Format Conversions and Query Rewriting for RDF* and SPARQL*

Jesper Eriksson
Amir Hakim

Supervisor: Olaf Hartig
Examiner: Olaf Hartig
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtidiga ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Jesper Eriksson & Amir Hakim
Abstract

*Resource Description Framework* (RDF) is a modern graph database model which has a proposed extension called *RDF*. However, currently there are no tools available that allow for transitions between RDF and RDF*. In this thesis tools were developed that allow for conversions between these two formats. A conversion tool for transforming RDF’s associated query language SPARQL into SPARQL* was also made. Furthermore, tools for converting between RDF* and another graph database model called *Property Graph* were developed. Measurements of the memory usage and conversion times for the developed conversion tools were made. Where possible, tests were also performed to control that there was no data loss in the conversions. The results showed varying memory usage and conversion times for the different conversions and that some conversions were very difficult to check for data corruption. Complex testing tools would be required to make sure all conversions are made correctly.
Acknowledgement

We would like to thank Olaf Hartig for offering us this thesis subject but also for the extensive help he has provided us during its course.
Table of Contents

1 Introduction .......................................................................................................................... 1
  1.1 Background ..................................................................................................................... 1
  1.2 Motivation ...................................................................................................................... 2
  1.3 Aim .................................................................................................................................. 2
  1.4 Delimitations .................................................................................................................. 3
  1.5 Research questions ......................................................................................................... 3

2 Theory .................................................................................................................................. 5
  2.1 Resource Description Framework .................................................................................. 5
    2.1.1 Turtle format ............................................................................................................. 5
    2.1.2 Reification statement ............................................................................................... 7
    2.1.3 SPARQL .................................................................................................................... 7
  2.2 Resource Description Framework* .................................................................................. 8
  2.3 Property Graph .............................................................................................................. 9
    2.3.1 Overview of GraphSON ........................................................................................... 10
    2.3.2 Overview of GraphML ............................................................................................. 10
    2.3.3 Overview of Comma-separated values ................................................................. 11
  2.4 Expressing RDF* as a PG .............................................................................................. 12
  2.5 Popular RDF frameworks in Java .................................................................................. 14
    2.5.1 Apache Jena ........................................................................................................... 14
  2.6 Testing .............................................................................................................................. 14

3 Method .................................................................................................................................. 15

4 Implementation & Conversion Algorithms ....................................................................... 17
  4.1 RDF to RDF* .................................................................................................................. 17
  4.2 RDF* to RDF .................................................................................................................. 18
  4.3 Serialization used for PG ............................................................................................... 19
  4.4 RDF* to PG .................................................................................................................... 20
  4.5 PG to RDF* .................................................................................................................... 21
  4.6 SPARQL* to SPARQL .................................................................................................... 22

5. Result .................................................................................................................................. 23
  5.1 Test setup: ....................................................................................................................... 23
  5.2 Conversion times ........................................................................................................... 24
  5.3 Memory usage ............................................................................................................... 28
  5.4 Validation of conversions .............................................................................................. 33

6. Discussion ............................................................................................................................ 35
  6.1 Results .............................................................................................................................. 35
6.1.1 Conversion times ........................................................................................................... 35
6.1.2 Memory usage ............................................................................................................. 36
6.1.3 Validation of conversions ............................................................................................ 36
6.2 Method .............................................................................................................................. 37
  6.2.1 Source criticism ......................................................................................................... 38
7 Conclusion .......................................................................................................................... 39
8 References ......................................................................................................................... 41
1 Introduction

1.1 Background

Databases in different forms have been used for a long time and the classic relational database model is currently the most common form [1]. Contrary to the name however, dealing with relationships is something relational databases are poor at [2] and in some cases, it is more useful to use a different kind of model to describe the data. Graph data modeling is a good alternative that can describe relationships between different data in a much easier way than a relational database is able to. Today there exists several different types of graph data models and the technology is widely implemented by tech giants such as Facebook and Google [2]. Two of the more dominant graph data models are Property Graph (PG) and Resource Description Framework (RDF).

A PG consists of vertices (also called nodes) that connect to each other through directed edges (also called arcs), these vertices and edges can be labelled as well as hold properties. Some variant of this model is used in the majority of the popular graph databases on the market [2]. Figure 1 demonstrates a simple property graph.

![Property Graph Diagram]

*Figure 1. The image demonstrates a simple property graph by introducing all the core elements.*

In the RDF model you have so called triples which consist of a subject, predicate and object and a single relationship direction between these. Developed by the World Wide Web Consortium (W3C) as a building block for the Semantic Web, the main use cases of this model are to interchange data over the web and to integrate data from different sources [3]. Figure 2 illustrates an RDF triple.
1.2 Motivation
Despite its strengths, the RDF model does however have some areas which need to be improved. In particular, writing something called statement-level metadata which can be described as making a statement about a whole triple in RDF. Statement-level metadata inflates the data significantly and makes queries much more complex [4]. There have been multiple approaches to solve this issue, but each of them had their own shortcoming which is why a new extension called RDF* was introduced by Hartig [4]. Its main feature is that the subject or object of a triple now can contain another triple by using a special syntax.

W3C also developed the officially associated query language for RDF which is called SPARQL. It too must be extended to SPARQL* to be able to handle data in the RDF* format.

There is currently a need to bridge the gap between RDF* and these other formats (PG, RDF and SPARQL) as the purpose of this extensions is to take a step in the direction of reconciling different data models with each other. Even though formal definitions for these conversions have been laid out by Hartig [5], actual tools to perform them do not exist.

1.3 Aim
The purpose of this thesis is to develop programs which can convert between different graph database formats. This achieves two things. First of all, it creates a way to implement and use RDF*, which is desirable since it is more readable for humans and has less data blow up if there exists statement-level metadata in the dataset than RDF. Second, this will further ease the exchange of information and data between sources which use different data graph models. Offering these conversion tools will allow the sources to keep their internal structure intact while at the same time being able to share data in a unified way through RDF*.

Figure 2. A visualization of a RDF triple.
1.4 Delimitations
There are many ways in which RDF can be serialized. The developed tools in this thesis are limited to handle Terse RDF Triple Language (Turtle) format for RDF files, Turtle* for RDF* files and CSV format for PG files.

The work in this thesis does not have any significant ethical aspects or societal links that are of importance, therefore there is no discussion of the work in wider context.

1.5 Research questions
By using the theory and formal definitions provided by Hartig’s research [5] as a foundation, we aim to answer the following questions.

1. Is it possible to develop a program in java that can convert in a streaming fashion between arbitrarily large datasets without loss of data:
   - From a RDF* file serialized in Turtle* to a RDF file serialized in Turtle and vice versa?
   - From a RDF* file serialized in Turtle* to a PG file serialized in CSV and vice versa?

2. Assuming the conversions are possible to implement, what kind of performance can be expected for each type of conversion on datasets of varying sizes in terms of conversion speed and memory usage?

3. Is it possible to develop a program in java that can convert from a SPARQL* query to an equivalent SPARQL query?
2 Theory
This chapter provides further necessary information regarding the different data models which this thesis handles. It also brings up existing tools which can be of use when working with these data models. This chapter also goes into limitations and possible loss of information during conversions. Finally, there is some information about testing.

2.1 Resource Description Framework
As described briefly in the previous chapter, the RDF model consists of triples. A triple which could also be referred to as a statement, contains three parts: the first part is called subject, the middle part is called predicate and the last part is called object [6]. Any of the individual parts can be referred to as a resource or a node. To get a better understanding of what a triple is, there is a need to describe a few terms. Therefore, the three existing node types are explained below.

- **Internationalized Resource Identifier (IRI)** is an identifier which can be used to mark any kind of thing, for example a resource on the Web.
- The term **literal** is a constant value of a specific type, commonly a string, a number, a Boolean or a date. A literal consists of several parts: the value represented as a string, its datatype and for strings an optional extra part called language tag.
- There is also something called **blank nodes**. Which does not refer to a specific resource but instead just provides information saying there is “some resource”.

The subject part of the triple can be of two different types, an IRI or a blank node. The predicate is always an IRI and the object can be any of the three different types, IRI, literal or blank node [6].

To elaborate even further what a triple is we can look at the subject as the node we speak about. The predicate is a property or relationship that the subject has. The object is a value for the property or a resource which the subject has a relationship to. For instance, the triple in Figure 2 would state “Dan has the age 29”.

The issue at hand with RDF which is of focus in this thesis lays with representing statement-level metadata. Currently it is expressed in a much more complex way than it is in a PG. This will be further explained in section 2.1.2.

2.1.1 Turtle format
RDF can be serialized in many different formats and one of these formats is called Turtle. It is a subset of the Notation3 (N3) format and one of its strengths is its very human-readable syntax. It is essential to understand the Turtle syntax to be able to later convert both to and from this syntax. Hence some of the most fundamental syntactic features of Turtle will be
explained here. To start with, there is something called prefix which is usually in the beginning of the file. The prefix is used to make it easier to read the file [6]. The prefix is just an alias for a long namespace IRI. The prefix is denoted as: @prefix “the alias” “namespace IRI”. In Example 1 there is a triple with turtle syntax which also has a prefix.

```
1 @prefix ex: <http://example.org/> .
```

*Example 1. A simple Turtle example containing a prefix and a triple.*

ex:Dan is shortened from <http://example.org/Dan>, ex:Sarah has been shortened in the same way. Meanwhile, the predicate does not use a prefix. In this example all the nodes are IRI’s, this is told by the nodes using aliases or the node being contained within “<>”. What this triple is saying is that “Dan knows Sarah”. The dot ‘.’ in the end means end of the triple. Example 2 further demonstrates the different kinds of triple separator symbols which exists in Turtle.

```
1 @prefix ex: <http://example.org/> .
2 ex:Dan ex:age 29 ;
3            ex:likes “Food” ,
4                            “ice-cream”.
```

*Example 2. Three regular triples expressed in Turtle.*

What this example is saying is that: “Dan has the age 29, Dan likes Food and Dan likes ice-cream”. As can be seen in the second line of the example, the line ends with a semicolon instead of a dot. This means that next triple will have the same subject as the triple before, thus making it necessary to only write out the next predicate and object. Line 3 ends with a comma which means that the next triple has the same subject and predicate as the previous triple [7]. The objects in these triples are all literals as none of them are contained within “<>” nor have a prefix attached to it. First object being an integer while the others are strings.

Whenever you have more than one triple which are not separated by a dot, this is called a triple block. For instance, Example 2 consists of one such block which has three triples.

The next essential thing regarding the Turtle syntax is the blank nodes. They are identified by the node always starting with “_:” followed by the blank node name, e.g. _:b1 could be a blank node. Putting it into context we could switch the current subject in Example 1 to the blank node _:b1 instead, which would give us the triple shown in Example 3 which states “Someone knows Sarah”.
Example 3. A simple Turtle example containing a prefix and a triple.

2.1.2 Reification statement
The last and perhaps most crucial thing is statement-level metadata. Statement-level metadata is statements about another triple. There is no efficient way to express this in RDF. Whenever we want to create such statement-level metadata we need to create a blank node (or an IRI) with the predicate `rdf:type` and the object `rdf:Statement` followed by another three triples having that blank node as subject and the predicates `rdf:subject`, `rdf:predicate` and `rdf:object`. The objects in these triples shall be the three elements of the triple we want to express our statement-level metadata about. These four triples together are referred to as a reification of a triple. In addition to this, the (metadata) statement that we want to make about the reified triple must be created as another triple with the same blank node (or IRI) as its subject or object. For example, to express that there is a probability of 50% that Dan knows Sarah, Example 1 would have to be extended further as demonstrated in Example 4.

```
@prefix ex: <http://example.org/> .
_:b1 <http://xmlns.com/foaf/0.1/knows> ex:Sarah .
```

Example 4. A reification and metadata about that triple.

Here it is displayed that we have introduced fives lines to just say one sentence about another triple, which is unnecessarily complex. The first line is the regular triple and next four lines are the reification statements and the last line’s predicate and object is in this case the metadata.

2.1.3 SPARQL
SPARQL is a declarative query language which is used to query RDF data. This language has a great number of features and it is therefore impossible to go into too much detail in this thesis. However, to get a feeling of how it looks we have provided an example query which could be run on the data in Example 2.

---

1 `rdf:` is a standard alias for the namespace http://www.w3.org/1999/02/22-rdf-syntax-ns#
Example 5. A basic SPARQL query.

The where clause of Example 5 contains “?x foaf:age 29” which is a triple pattern. The difference between a triple pattern and a regular triple is that any of the three elements in a triple pattern can be a variable [8]. In SPARQL a variable is denoted by a question mark “?” before the variable name. A set of triple patterns together constitute a basic graph pattern.

This specific example query is going to return all subjects which has the age 29. In our simple example it would just be <http://example.org/Dan>.

2.2 Resource Description Framework*

RDF* is just a minor modified version of the original RDF model. It has a different way to express statement-level metadata which is easier to read. This is done by creating a triple from the reification statement and putting it as the subject or object of a triple. Specifically, the objects of the triples with the predicates rdf:subject, rdf:predicate and rdf:object are used. In other words, an entire triple is now stored as single node as the subject or the object of a new triple. Apart from this RDF* has the same functionality and is expressed the same way as RDF.

To work with RDF* files, a modified version of Turtle called Turtle* was introduced [5]. Expressing the Turtle statements in Example 4 as Turtle* would result in Example 6.

Example 6. Equivalent to Example 4 but expressed in Turtle*.

Everything inside the “<<” and “>>” tags can be viewed as a single node and will be referred to as a nested triple in this paper. In Example 6 the nested triple is set as the subject of a triple, however, it can also be the object (but never the predicate). It is also possible for a nested triple to contain another nested triple. As seen in Example 6 the statement only requires one line in Turtle* instead of five to express the same thing in Turtle. It is quite apparent that it is much easier to express metadata in this way as it excludes the use of reifications.

Mapping from RDF* to RDF is a simple task as a normal reification statement is created from the nodes in the nested triple. A blank node id is also generated in this process.
With the introduction of RDF* and Turtle* there was a need to extend the SPARQL query language into SPARQL*. Specifically, it needs to support triple patterns that are nested. An example of such query is displayed in Example 7.

```
1 prefix ex: http://example.org/
2 SELECT ?x ?name ?src WHERE{
  <<?x foaf:knows ?name>> foaf:from ?src . }
```

*Example 7. A SPARQL* query with nested triple pattern.*

Furthermore, the `BIND` keyword (which makes it possible to bind a value to a variable) needed to be able to bind nested triple patterns with variables as well [5]. Example 8 shows how a BIND query can look.

```
1 prefix foaf: http://xmlns.com/foaf/0.1/
2 prefix ex: http://example.org/
3 SELECT ?c WHERE {
  4 BIND( <<?s foaf:knows ?p>> AS ?t )
  5 ?t ex:likes ?c .}
```

*Example 8. A SPARQL* query using the `BIND` keyword.*

2.3 Property Graph

There are many different informal descriptions of the property graph model. This paper will follow the descriptions given by Robinson et al [2] since Hartig’s research is based on them as well. Those descriptions state that the labelled property graph model should have the following features:

- “A labelled property graph is made up of nodes, relationships, properties, and labels.
- Nodes contain properties. Think of nodes as documents that store properties in the form of arbitrary key-value pairs.
- Nodes can be tagged with one or more labels. Labels group nodes together, and indicate the roles they play within the dataset.
- Relationships connect nodes and structure the graph. A relationship always has a direction, a single name, and a start node and an end node—there are no dangling relationships. Together, a relationship’s direction and name add semantic clarity to the structuring of nodes.
- Like nodes, relationships can also have properties. The ability to add properties to relationships is particularly useful for providing additional metadata for graph algorithms, adding additional semantics to relationships (including quality and weight), and for constraining queries at runtime.” [2]
Currently there is no official standard for how to serialize a property graph [9] [10] [11]. One of the major actors when it comes to representing and working with graph data is Neo4j. They use their own interpretation of comma-separated values (CSV) as their formalization format [12]. Another example of someone using their own interpretation of the CSV format is Amazon with their Amazon Neptune [13].

Another big actor in this field is Apache TinkerPop™, they provide an open source framework for working with graph databases. Their documentation mentions the formats GraphSON, GraphML and Gryo [14]. The latter mentioned is a binary graph serialization format, and since this paper focuses on text based serializations, it doesn’t delve deeper into the Gryo format.

Any of these formats can be used, but some are more appropriate than others. Desirable characteristics in the format is low complexity as well as its ability to be streamed, meaning it won’t be necessary to load the entire file into memory to preserve the integrity of the data.

### 2.3.1 Overview of GraphSON

There exist several different versions of GraphSON and they are not backwards compatible. GraphsSON 3.0 is the most recent version and was deployed in TinkerPop 3.3.0 (current version of TinkerPop is 3.3.2). This format uses an adjacency list, meaning each line in a GraphSON file represents a single vertex. Each line also contains the properties and labels that the vertex has, as well as all incoming and outgoing edges with their respective labels and properties. Example 9 shows how one of these lines can be represented. When the data is expressed in this manner the whole document is not valid JSON, but each line separately is. It is possible to wrap the whole adjacency list to make it a valid JSON file, but then you lose the ability to split or stream the file.

```
1 {"id":1,"label":"human","inE":{"knows:[{"id":4,"outV":2,"properties":{"certainty":0.7}}],
"outE":{"created:[{"id":5,"inV":3,"properties":{"certainty":0.8}}],
"properties":{"name:[{"id":6,"value":"alice"}],"age:[{"id":7,"value":29}]}}
```

*Example 9. The following example shows a one row excerpt from a GraphSON file.*

### 2.3.2 Overview of GraphML

This format is XML-based and one of its main predecessors is Graph Modeling Language (GML). It uses a classic XML tree structure with tags such as <graph>, <node> and <edge> just to name a few as in shown in Example 10 [15].
2.3.3 Overview of Comma-separated values

As previously mentioned, there are different interpretations for how the CSV format should look. We chose to focus on Amazon Neptune’s interpretation as it has good documentation and it follows the RFC 4180 CSV specification [13][16] which helps making the format requirements even more clear.

The CSV format is basically a table that is written from the top down and from left to right where each cell is separated by a comma. Each row represents an entry and the columns define different values that entry might hold. With Amazon Neptune’s interpretation of CSV, you divide the property graph in two csv files, the vertex file and the edge file. The vertex file contains all the vertices and their attributes whereas the edge file contains all the edges between those vertices and any properties those edges might have. In the vertex file it is possible for multiple values to be entered into a single cell as long as they are separated by a semicolon.

Both files always start with a header row (which differs in the two files).
In the vertex file the header must contain a vertex-id column, a label column and all the property labels as individual columns. The datatype of the properties has to be specified in the header as well. Example 11 is an example of what a vertex file can look like.

```
1 id, name: String, born: Date, age: Int, label
2 v1, Alice, 1998-10-05, 20, Person
3 v2, Bob, , 21, Person
```

*Example 11. A vertex file that follows the Amazon Neptune standard.*
In the edge file the header must contain an edge-id column, a to and from column respectively, a label column and any potential property labels of the edges as individual columns. The datatype of the properties need to be specified in the edge file as well. Example 12 demonstrates an edge file.

<table>
<thead>
<tr>
<th>id, from, to, label, certainty:Double, created:Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1, v1, v2, knows, 0.8,</td>
</tr>
<tr>
<td>e2, v3, v6, knows,, 2018-04-02</td>
</tr>
</tbody>
</table>

Example 12. An edge file that follows the Amazon Neptune standard.

2.4 Expressing RDF* as a PG

Due to RDF being more expressive than PG there are some problems that must be addressed. Hartig [5] brings up two different ways in which RDF* can be conveyed as a PG and the flaws of each method:

1. The first way is to take the subject and object of a triple and create a vertex for each, then the corresponding predicate is used to create the directed edge that connects those vertices. The vertices then always contain at least two properties, one which specifies whether it is a literal, IRI or blank node and one that states the value of said type. If the object was a literal, the vertex will receive a third property which states the datatype of the literal. Additionally, if the datatype was a string with a language tag, the vertex will receive a fourth property called language which saves the language tag. Any statement-level metadata about that triple is expressed in the properties of the edge for that relationship. Figure 3 gives an example of what this could look like in a PG.

The problem with this method is that the object of a statement-level metadata triple can be an IRI in RDF*, something that cannot be expressed in a PG since the value of an edge property cannot be another vertex. For instance this triple cannot be expressed with this method:

```
```

whereas this can:

```
```

Figure 3. A visualization of the first way to express RDF* as a PG.
2. The second way is to first divide all triples into relationship triples and attribute triples. Relationship triples must have IRIs or blank nodes as both their subject and object whereas attribute triples must have a literal as the object. The relationship triples are then used to create all vertices and edges while the attribute triples are turned into properties of the corresponding vertices.
Statement-level metadata is transformed to edge properties. Figure 4 demonstrates this method in a simple PG.
The problem with this method is that statement-level metadata about attribute triples cannot be expressed. This is because one cannot give the property of a vertex or edge their own set of properties. For example this triple cannot be expressed with this method:
<<ex:bob foaf:age 29>> ex:certainty 1.0.

![Figure 4. A visualization of the second way to express RDF* as a PG.](image)

As a countermeasure to these limitations, Hartig proposes a set of rules which allow for a lossless transformation used in combination with the first method expressed above. The rules demand that:
1. “Metadata triples are not nested within one another.
2. Metadata triples embed triples as their subject only (not as their object).
3. The object of any metadata triple must be a literal.
4. For any literal in the triples it must be possible to convert the literal to a data value.”

Since one of the strengths of RDF* is the way it handles statement-level metadata, it would be counterproductive to convert it to PG in a way that specifically loses this kind of information. With this in mind, it is our assessment that the best choice is to work with the first method of expressing RDF* as a PG combined with the four rules stated above. This would make it a lossless conversion.

It’s also worth noting that in Hartig's research it is assumed that the vertices in PGs don’t have any labels, so for simplicity’s sake the same assumption is made in this thesis.
2.5 Popular RDF frameworks in Java

There are several available libraries dedicated to working with RDF graphs. They help with processing and working with RDF data in many ways. For example, they make it possible to parse many different RDF formats, to load and compare different RDF graphs, to extract or modify certain data, to transform the data into another RDF format etc. This is of course of high value for a thesis like this. One of the more established libraries in this domain is Apache Jena.

2.5.1 Apache Jena

Apache Jena, also referred to as Jena, is an open source framework which is one of the most popular in regard to graph database tools [1]. Jena has a very wide set of functionalities in regard to RDF. For instance, it supports most of the different RDF formats and it can load a whole RDF file into a Model object which is then easy to work with. It can also read triples in a streaming fashion and also has the possibility to compare whole triples or parts of triples with other triples which is quite handy. It also keeps track of the resource type of a subject, predicate or object. This is just a few of Jena’s functionalities. However, what it does not have is a parser for RDF*, which means some modifications of the Jena library is necessary to be able to use it when converting to/from RDF*. An alternative approach to extending Jena is to build a custom parser from the ground up.

2.6 Testing

There are two types of tests that need to be performed, one is performance tests and the other is corruption tests.

When testing for performance the focus will be on looking at how much time a conversion takes as well as the memory used by the program during that conversion.

The point of corruption testing is to verify that the dataset before and after conversion still equal the same graph. In the case of converting SPARQL* to SPARQL, the desirable outcome is a query that would return the same data as the SPARQL* query.
3 Method

This chapter describes the methods and the process that we have used to address the research questions and to achieve the aim of our thesis project.

Initially Hartig’s papers on RDF* [4][5] were examined to get an understanding on what RDF* accomplishes in comparison to regular RDF and how it works in general. A large amount of time was also spent on researching the official RDF documentation published by W3C. This was necessary to be able to learn the syntax of the Turtle format and to get an understanding of how RDF is built up.

Once enough information was gained about RDF and RDF*, focus was shifted to gathering information about PGs. Since this field is not as well defined with official standards like RDF, there is no clear answer to which formats to use and which frameworks might be helpful. Therefore, a lot of time was spent exploring different PG serialization formats and finding out if they work well with RDF* and streaming.

SPARQL* and SPARQL were also studied just enough to get a grasp of what a SPARQL* query should look like after being transformed to SPARQL.

With enough information and understanding about all these formats, an evaluation was made to figure out the most beneficial order to develop the different conversion programs. It was concluded that conversions between RDF and RDF* (both ways) should be developed first since both formats are well documented and therefore easier/faster to implement. This should be followed by developing conversions that handle PGs. SPARQL* to SPARQL conversions had the lowest priority since currently no database management program can execute SPARQL* queries.

Before going fully into the development phase, the frameworks Jena and RDF4J were both briefly examined to find out which would be easiest to implement. Because our supervisor had previous knowledge of Jena and RDF4J isn’t used to the same extent [1], Jena was chosen to assist us in our conversion programs. The idea of creating a custom parser was also explored but it very quickly became obvious that this would be too large of a task. It was instead decided that extending Jena would be a better route to take.

At this point the development phase began. RDF to RDF* was developed first followed by RDF* to RDF. Afterwards, the PG to RDF* and RDF* to PG conversion programs were developed. Finally, SPARQL* to SPARQL conversions were made possible.

Once all the programs had been developed, testing for performance and for possible corruption began. To measure the elapsed time of a conversion the Stopwatch API of the
Google Guava library was used. For measuring the memory usage of the program, Java’s built in class Runtime was sufficient.

Checking for corruption turned out to be a very complex task and we were only able to control two scenarios. One test performed was converting from RDF to RDF* and then back to RDF again and then with the help of Jena compare the original with the resulting graph for isomorphism (i.e. check if the two graphs have the same form). The other test was converting from RDF* to RDF and back to RDF* and comparing them in the same manner as before.
4 Implementation & Conversion Algorithms

This chapter goes more in-depth into how the actual implementation was performed. There is also some high-level pseudo code explaining each conversion.

4.1 RDF to RDF*

The first conversion to be implemented was the RDF to RDF* where Jena version 3.7.0 was used (this version was the only one used in this thesis). With the help of Jena’s PipedRDFIterator<Triple>, PipedRDFStream<Triple> and RDFParser objects, triples were read into the program in a streaming fashion. The reason for streaming was due to the possibility that the data files can be very large (several gigabytes in size) and would risk overflowing the memory if they were to be fully loaded into the memory.

A crucial fact is that the RDF file would always have to be read twice. During the first reading, all reification statements were separated out and saved in the memory. This was because metadata about a reified triple can exist both before and after the corresponding reification statements in the file. If the case would be that the metadata statement existed before the reified statement and the file was read just once, the information that it was metadata would be lost (in other words, it would be regarded as a standalone triple). During the first reading, all prefixes were also read and stored into memory.

During the second streaming of the file the program reads and prints one triple at a time. All regular statements (i.e. no reification statements and no metadata statements) were simply printed out as they were read as no modification was needed for those. Meanwhile, whenever a metadata statement was read, a recursive method was called to nest the statement (as many layers as necessary) according to RDF* syntax before printing it. Any reification statements were ignored during the second reading as these didn’t need to be processed further.

When printing regular Turtle triples, Jena has an interface called NodeFormatter which formats Nodes according to a specified format, for instance Turtle. However, to be able to print to the Turtle* format, it was necessary to extend the interface to handle this. This was done by our supervisor Olaf Hartig and the extension is available in the public GitHub repository RDFstarTools [17]. The extended NodeFormatter is called NodeFormatterTurtleStarExtImpl. As the formatter only formats nodes, a method had to be developed to format the output into Turtle* blocks (see definition for block in section 2.1.1) when possible, i.e. whenever triples with the same subject were read consecutively. Pseudo code for the implementation is shown in Figure 5.
Read the entire input file twice (one triple at a time).
First lap:
Store all prefixes in a map. Also, store all information about each reification statements in a
different map together with its subject as a key.
Second lap:
If read triple is part of reification statement, do nothing.
Else if read triples subject or object exists in the reification map they need to be nested. The
nesting is done by calling a recursive function. The recursive function uses the information in
the reification map to nest as many levels as needed. Finally, print the transformed triple.
Else if read triple was a regular triple, print out as it is.

Figure 5. Pseudo code for the RDF to RDF* conversion.

4.2 RDF* to RDF

The second conversion to be implemented was the RDF* to RDF. As previously mentioned,
Jena was lacking the ability to read RDF* files or more specifically Turtle* files. However, our
supervisor Olaf Hartig once again made this possible by extending the Jena library with such
a parser and adding it to the RDFstarTools repository.

Just like in the RDF to RDF* program the triples were read in a streaming fashion to avoid
memory issues. In this program the file was read twice as well. The first reading of the file
was just to gather all prefixes. This might seem a bit excessive but was necessary due to
limitations in how Jena chooses to store prefixes in the memory. When Jena reads a row
from a file and it detects a prefix, it automatically adds it to an internal map. This internal
map doesn’t allow one to retrieve the most recently added prefix, instead it only returns the
entire map for printing. This would lead to printing many duplicates of the prefixes.

In the second reading each triple is read and printed one at a time. Every triple’s nodes were
checked to see if they contained a nested triple. If the node was nested, a recursive function
to unnest the node completely was called. For each layer of nesting a reification had to be
created and printed to the output file. The blank node id of these reifications was linked to
its respective nesting and stored in memory for future reference. Furthermore, any regular
triples (i.e. they contained no nestings) encountered in the second reading were simply
printed out in the same format as they were read. Printing was done using Jena’s regular
turtle parser with the NodeFormatter. Pseudo code for the implementation is shown in
Figure 6.
Read the entire input file twice (one triple at a time).
First lap:
Store all prefixes in a map.
Second lap:
If read triple is contains nesting, use recursive function to unnest as many levels as needed.
Create and print reification (and the associated metadata triple) for every unique unnested triple. Also, store the nested triple as the key with a generated blank node id as value in a map.
This map is used to refer reoccurring nestings to the same blank node id and to avoid printing the corresponding reification triples more than once.
Else if read triple was a regular triple, print out as it is.

Figure 6. Pseudo code for the RDF* to RDF conversion.

4.3 Serialization used for PG
In the Theory chapter several PG formats were presented. However, they all have pros and cons which this section covers.

Starting with GraphSON the main problem occurs when converting from RDF* to PG. To be able to create a vertex row, all information about that particular vertex is needed. This means any information about incoming and outgoing relationships with that particular triple’s subject has to be located in the RDF* file. The information gathering process would then have to be repeated for each new vertex row as well. This would be extremely inefficient and is not appropriate for streaming. For this reason, it was decided to not work with GraphSON in this thesis.

Early on into the research of GraphML some key problems were discovered. First of all, according to the paper by Tomaszuk [10] some of the characters used in RDF are not allowed in XML attributes (it is however unknown to us whether this issue has been resolved since that publication). Second, it appears this format has not received any updates in a long time as the latest news from the official webpage is from 2007 [18]. Finally, the nested structure of this format as well as its complex expressiveness makes it little bit more difficult to work with in a streaming fashion. Because of these reasons this format was deemed unfit for this thesis and no further research was made about the format.

As the Amazon Neptune CSV format is specifically made for handling property graphs it definitely has a lot of pros, however, it does have a minor problem when trying to convert to RDF*. The “Date”-type in CSV isn’t always compatible with the equivalent RDF type (and by extension RDF*). RDF defines this way of writing YYYY-MM-DD as “Date” and this way YYYY-MM-DDTHH:mm:SS as “DateTime”, whereas CSV would define both simply as “Date”. Furthermore, CSV also allows for “Date” types to be written as YYYY-MM-DDTHH:mm which is not supported in RDF. In this case one can simply add the value “00” for the seconds to
transform it to “DateTime”. This can possibly lead to minor distortions of the data when transforming from PG to RDF*.

This format is quite suitable when converting from PG to RDF*, but when converting in the other direction things have to be done differently to keep the promise of a lossless conversion (as mentioned in section 2.4). Therefore, when converting from RDF* to PG the header in the vertex file would have to be changed according to Example 13 but the edge file can keep the same format.

<table>
<thead>
<tr>
<th>ID, KIND, IRI, LITERAL, BLANK, DATATYPE</th>
</tr>
</thead>
</table>

*Example 13. The header in the vertex file when outputting from RDF* to PG.*

Since there is no way to store the prefixes used in RDF* in a PG these will be lost in a conversion. Any prefixes used in triples will therefore have to be transformed to contain the full IRI before a conversion to PG.

Despite only being able to convert one way using these two different CSV formats this formatting was considered better than GraphSON and GraphML and therefore chosen for this thesis.

4.4 RDF* to PG

In this conversion the extended Jena parser was used again to perform one single reading of the file with one triple being read at the time. However, in this conversion almost all the information had to be saved into memory. All predicates had to be known to be able to create the headers in the CSV files. Before printing a row in the edge file, every single metadata statement about that particular relationship had to be found and stored in order to be written in the correct column. Also, every subject and object had to be stored in a map together with a unique id as this was necessary to create the connection between two vertices in the edge file.

The read triple’s information was categorized as either vertex data or edge data. All subjects and objects in regular triples were considered vertex data and all predicates and metadata were considered edge data. Once all data had been saved and organized in memory, one row was printed at the time into the vertex file. Finally, one row at the time was printed into the edge file. Pseudo code for the implementation is shown in Figure 7.
Read the entire input file once (one triple at a time).
If read triple is a metadata triple, store the subject and object of the nested triple in a vertex map with individual ids assigned. Save entire nested triple in a map as the key together with a pair of the predicate and all its metadata. Also save the predicate in a list that stores all headers for the edge file.
If read triple is a regular triple simply store subject and object in vertex map with individual ids assigned.
Use vertex map to print all vertices to the vertex file.
Use edge header list to print all headers to edge file.
Use edge map to print all edges with corresponding metadata to the edge file.

Figure 7. Pseudo code for the RDF* to PG conversion.

4.5 PG to RDF*
The third conversion to be implemented was the PG to RDF*. Here a parser provided by the Apache Commons CSV library version 1.5.1 was used for reading. In a PG the equivalent of a RDF* predicate is the edge label. In PG this is always a string, but in RDF* the predicate is required to be a IRI. Therefore, an optional extra input file was allowed in which the user could connect specific header strings with prefixes and their IRIs. The program started by reading this file (if it was given) and stored the information to be used later when printing to the Turtle* file. Example 14 demonstrates how the content of a prefix file could look.

| 1 name http://xmlns.com/foaf/0.1/ foaf: |
| 2 age http://xmlns.com/foaf/0.1/ foaf: |
| 3 creationDate http://example.org/ |

Example 14. Contents of the optional prefix file. In this example any CSV header with the value age will be converted to the predicate foaf:age in the Turtle* file. Any header containing creationDate will become http://example.org/creationDate.

Next, the program reads the vertex file one row at the time and processing one cell at a time. For each row a blank node named after the vertex id is generated. This blank node becomes the subject for all triples created from this row. Every cell containing a value is converted to a triple using the cell value as the object and the header of that column as the predicate. The data type also specified in the header of each column is used to format the object and any prefixes are attached if available. After this, the information is printed as a triple to the output file, meaning there never will be large amounts of data stored in the memory.

Once the vertex file has been dealt with, the edge file is read row by row and processed in a similar fashion. A triple is created for each row in the edge file using the values in the “from”, “to” and “label” columns as the subject, object and predicate respectively. Any cells containing data with a different header than the ones just mentioned represent metadata.
These are used as the objects in the metadata triples. Finally, the triples are printed to the same output file as before. Pseudo code for the implementation is shown in Figure 8.

| Read prefix file (if specified) and store in a map, refer to this map later when formatting triples.  
Read the entire input vertex file once (one row at a time).  
First row read is header, save name together with datatype in a list.  
For every other row, create a triple for each cell with a value on the row.  
Format and print triples to result file.  
Read entire input edge file once (one row at a time).  
Create triple for each edge and metadata triples for any edge labels. Format and print triples to previous result file. |

Figure 8. Pseudo code for the PG to RDF* conversion.

4.6 SPARQL* to SPARQL
The last conversion implemented was SPARQL* queries to SPARQL queries. In this area our supervisor Olaf Hartig helped by implementing a SPARQL* parser by extending the Jena library once again. Once the queries could be read, the nested triple patterns had to be unnested and replaced by reification statements in similar way to how it was done in the RDF* to RDF conversion. This was done by overriding two of the transform() methods in the ElementTransformCopyBase superclass. The transform method normally works as a logical expression optimizer in Jena, but we overload it with the purpose of replacing nested triple patterns. After this the new modified query could be printed out. Pseudo code for the implementation is shown in Figure 9.

| Read the entire input file once (one triple pattern at a time).  
If triple pattern contains nesting, recursively unnest until no more nestings exist.  
Create reification statement for each unnesting and add this to a data structure for printout. Store each nested triple as key together with the reifications generated blank node id. This map is checked each time to refer to the correct blank node id in case the same nesting reoccurs again. |

Figure 9. Pseudo code for the SPARQL* to SPARQL conversion.
5. Result

\textit{In this chapter the results for the different tests that were run on all the conversions will be presented and explained.}

5.1 Test setup:

All the tests were run on a pc with the following specifications:

\begin{itemize}
\item \textit{Intel i5 4690k 4.2 Ghz},
\item \textit{8 GB 1600 Mhz ram},
\item \textit{120 GB SSD drive},
\item \textit{Windows 10}
\end{itemize}

The results from the performance and corruption tests are divided into the three categories: conversion times, memory usage, and validation of conversions. The conversion times tests are meant to achieve an understanding of how the conversion tools behave in terms of times required to perform their conversions. Similarly, the memory usage tests are meant to obtain an understanding of how the conversion tools behave in regard to required memory when performing their conversions. Finally, the validation of the conversion tests is meant to control the integrity of the output file.

All the different datasets used in the tests are listed in Table 1. Each test was repeated five times and then the average was calculated and used in the graphs. File 1 and File 2 were also divided into smaller files: a half, a quarter, and an eighth of its full size, where each was tested five times separately. When calculating the standard deviation, Excel’s built in function \textit{STDEV.S} was used.

File 1 was used in the RDF to RDF* conversion. File 2 was used in the RDF* to RDF conversion as well as RDF* to PG. File 5 to 12 were used in the PG to RDF* conversion. File 3 and 4 were both used in the RDF to RDF* as well as the RDF* to RDF conversion.
<table>
<thead>
<tr>
<th>File name</th>
<th>Format</th>
<th>File size, MB</th>
<th>Content</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>File 1</td>
<td>Turtle</td>
<td>453.73</td>
<td>~2.1M reification statements and metadata statements about the corresponding reified triples</td>
<td>Yago dataset²</td>
</tr>
<tr>
<td>File 2</td>
<td>Turtle</td>
<td>245.94</td>
<td>~2M nested triples with one level of nesting</td>
<td>Yago dataset</td>
</tr>
<tr>
<td>File 3</td>
<td>Turtle</td>
<td>161.39</td>
<td>No reifications/nestings</td>
<td>DBpedia³</td>
</tr>
<tr>
<td>File 4</td>
<td>Turtle</td>
<td>1094.95</td>
<td>No reifications/nestings</td>
<td>DBpedia</td>
</tr>
<tr>
<td>File 5 &amp; 6</td>
<td>CSV</td>
<td>2.068 &amp; 33.696 (tot 35.76)</td>
<td>~27K vertices, ~565K edges</td>
<td>LDBC Social Network Benchmark (SNB)⁴</td>
</tr>
<tr>
<td>File 7 &amp; 8</td>
<td>CSV</td>
<td>0.822 &amp; 10.668 (tot 11.49)</td>
<td>~11K vertices, ~180K edges</td>
<td>LDBC Social Network Benchmark (SNB)</td>
</tr>
<tr>
<td>File 9 &amp; 10</td>
<td>CSV</td>
<td>0.293 &amp; 2.614 (tot 2.907)</td>
<td>~3.5K vertices, ~45K edges</td>
<td>LDBC Social Network Benchmark (SNB)</td>
</tr>
<tr>
<td>File 11 &amp; 12</td>
<td>CSV</td>
<td>0.128 &amp; 0.816 (tot 0.944)</td>
<td>~1.5K vertices, ~14K edges</td>
<td>LDBC Social Network Benchmark (SNB)</td>
</tr>
</tbody>
</table>

Table 1. Dataset files used for testing.

5.2 Conversion times
The elapsed time during each conversion was measured using the Stopwatch API of the Google Guava library which measured the time from the beginning of the program until the program had finished executing.

In the Graph 1 and Graph 2 below the number of metadata triples are shown on the x-axis and the conversion times are listed on the y-axis.

Graph 1. The RDF to RDF* relationship between conversion times and number of metadata triples. The datasets tested in this graph were File 1 and subsets of that file.

³ http://downloads.dbpedia.org/2016-04/
As seen in Graph 1 the conversion time is increasing in a seemingly linear fashion with the number of metadata triples. The standard deviation is very small.

Graph 2. The RDF* to RDF relationship between conversion times and number of metadata triples. The datasets tested in this graph were File 2 and subsets of that file.

Graph 2 shows a very similar pattern as in Graph 1, but the standard deviation is slightly higher.

In the graphs below the conversion times for the different conversions and different files are presented. The x-axis shows the file size in Megabytes and the y-axis shows the conversion time in seconds.
Graph 3. The RDF to RDF* conversion times. The datasets tested in this graph were File 1 and subsets of that file.

As seen in Graph 3 the conversion time is increasing approximately in a linear fashion with the file size. The standard deviation is increasing as the file size grows but is still relatively small.

Graph 4. The RDF* to RDF conversion times. The datasets tested in this graph were File 2 and subsets of that file.
In Graph 4 similar results as Graph 3 are shown, i.e. the time seems to be linear with the increasing file size. Graph 4 shows the same pattern for standard deviation as Graph 3.

**Graph 5.** The PG to RDF* conversion times. The datasets tested in this graph were the paired files File 5 & 6, File 7 & 8, File 9 & 10 and File 11 & 12.

Graph 5 shows a conversion time which appears to be fairly linear to the file size. In regards of standard deviation, it seems to follow the same pattern as before.

**Graph 6.** The RDF* to PG conversion times. File 2 and subsets of it were tested in this graph.
Graph 6 is looking very similar to the previous graphs both regarding linearity as well as the standard deviation.

Table 2 contains trend lines for Graph 3-6 where the slope value is of interest. It shows that the PG to RDF* has by far the highest conversion rate, these results will be thoroughly discussed in the next chapter.

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Graph</th>
<th>Trend line equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDF to RDF*</td>
<td>Graph 3</td>
<td>$y = 0.0873x + 0.7412$</td>
</tr>
<tr>
<td>RDF* to RDF</td>
<td>Graph 4</td>
<td>$y = 0.1566x - 1.2715$</td>
</tr>
<tr>
<td>PG to RDF*</td>
<td>Graph 5</td>
<td>$y = 0.0292x + 0.1225$</td>
</tr>
<tr>
<td>RDF* to PG</td>
<td>Graph 6</td>
<td>$y = 0.0907x - 0.354$</td>
</tr>
</tbody>
</table>

Table 2. The trend lines for conversion speed in relation to file size.

5.3 Memory usage
The memory usage was calculated by using Java’s `Runtime` class which calculates the memory used by the Java Virtual Machine [19]. The numbers were retrieved in the following way:

1. At start of the program the total memory is subtracted by the free memory to get the current memory used.
2. Right before ending the program, Java’s garbage collector is executed to collect any garbage in the memory. This is followed by calculating the current memory used in the same way as step 1.
3. The difference between the value at the start and the end is calculated. That result is the actual memory usage of the conversion program.

Graph 7 and Graph 8 display the number of reified/nested triples that need to be stored in memory as the size of the test files used increase. On the x-axis the file size in Megabytes is shown and on the y-axis the number of reified/nested triples which are stored in the programs memory are shown.
Graph 7. The RDF to RDF* relationship between number of saved reified triples in memory and the file size. The datasets tested in this graph were File 1 and subsets of that file.

Graph 7 exhibits a linear pattern, meaning the bigger files have a linearly increasing amount of content that needs to be stored in memory when running the conversion.

Graph 8. RDF* to RDF relationship between number of saved nested triples in memory and the file size. The datasets tested in this graph were File 2 and subsets of that file.

Graph 8 displays the same pattern as in Graph 7, i.e. as the file size expands the number of nested triples that need to be stored in memory increase in a linear fashion.
In the graphs below the memory usage of the conversion programs for the different files are presented. On the x-axis the file size in Megabytes are shown and on the y-axis the memory usage in Megabytes are shown.

![Graph 9. RDF to RDF* memory usage during conversion. The datasets tested in this graph were File 1 and subsets of that file.](image)

As shown in Graph 9 the memory usage is increasing linearly with the file size. The standard deviation is close to non-existent at all times.

Two tests with two RDF datasets containing no reifications were also performed in the RDF to RDF* conversion to see how the content of the file could affect the memory usage. This is displayed in Table 3 where we see that the memory usage is close to zero when there are no reification statements as no data needs to be saved.

<table>
<thead>
<tr>
<th>File name</th>
<th>File size, MB</th>
<th>Memory usage, MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>File 11</td>
<td>161.4</td>
<td>~0</td>
</tr>
<tr>
<td>File 12</td>
<td>1094.954</td>
<td>~0</td>
</tr>
</tbody>
</table>

*Table 3. Shows the memory used when converting a file with no reifications from RDF to RDF*. 
Graph 10. RDF* to RDF memory usage during conversion. The datasets tested in this graph were File 2 and subsets of that file.

In Graph 10 we see a similar behaviour as in Graph 9. It seems to be linear and the standard deviation acts in the same manner as well.

Two tests with two RDF datasets without any nestings were also performed in the RDF* to RDF conversion to see how the content of the file could affect the memory usage. This is displayed in table 4 which shows the memory usage is close to zero when there are no nestings.

<table>
<thead>
<tr>
<th>File name</th>
<th>File size, MB</th>
<th>Memory usage, MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>File 11</td>
<td>161.4</td>
<td>~0</td>
</tr>
<tr>
<td>File 12</td>
<td>1094.954</td>
<td>~0</td>
</tr>
</tbody>
</table>

Table 4. Shows the memory used when converting a file with no nested triples from RDF* to RDF.
Graph 11. PG to RDF* memory usage during conversion. The datasets tested in this graph were the paired files File 5 & 6, File 7 & 8, File 9 & 10 and File 11 & 12.

Graph 11 shows a zero-memory usage. This is due to the program not saving any information permanently in the memory during execution. A small amount of memory will however be used temporarily but is cleaned up by the garbage collector during runtime. The standard deviation is extremely close to zero at all times.

Graph 12. RDF* to PG memory usage during conversion. File 2 and subsets of it were tested in this graph.

Graph 12 shows a linear pattern. The standard deviation is similar to the previous graphs.

Table 5 contains trend lines which shows memory usage efficiency for Graph 9-12 where the slope value is of interest. It shows that the PG to RDF* is unparalleled in terms of memory usage, these results will be thoroughly discussed in the next chapter.
Conversion | Graph     | Trend line equation
---|---|---
RDF -> RDF* | Graph 9 | \( y = 1.3891x - 7.1265 \)
RDF* -> RDF  | Graph 10 | \( y = 3.8541x - 15.914 \)
PG -> RDF*   | Graph 11 | \( y = 0 \)
RDF* -> PG   | Graph 12 | \( y = 3.0388x - 3.8474 \)

Table 5. The trendlines for the memory usage of the conversion programs.

5.4 Validation of conversions

Jena’s method `isIsomorphicWith` is used to compare the graphs. Tests were run on File 1, File 2 and two small files (one Turtle file and one Turtle* file) created by us which covered some of the very many different cases that might occur.

The result from running the isomorphic test on File 1 (RDF to RDF* back to RDF) returned false, meaning the conversions introduced some kind of change which resulted in a new and different graph than the original. This is simply caused by the fact that the dataset in File 1 contains an unknown amount of illegal statements to begin with. An example of such illegality which exists in File 1 is demonstrated in Example 15.

```
1 _:id_10000z6_1ia_1677krt rdf:type rdf:Statement ;
2                                               rdf:subject <Gmina_Obrzycko> ;
3                                               rdf:predicate rdfs:label ;
4                                               rdf:object "Obrzycko" .
5 _:id_10000z6_1ia_1677krt rdf:type rdf:Statement ;
6                                               rdf:subject <Gmina_Obrzycko> ;
7                                               rdf:predicate rdfs:label ;
8                                               rdf:object "Obrzycko"@fra .
```

Example 15. Two different reifications using the same blank node id.

In RDF it is not allowed for the same blank node id to be used as subject in two different reification statements. In Example 15 the blank node "_:id_10000z6_1ia_1677krt" is referencing two different triples, `<Gmina_Obrzycko> rdfs:label "Obrzycko"` as well as `<Gmina_Obrzycko> rdfs:label "Obrzycko"@fra` at the same time. Since the converted file is free of these errors, they cannot ever be isomorphic. Correcting these incorrect statements manually is simply too time consuming as the file has four million rows.

When running the isomorphic test on File 2 (RDF* to RDF back to RDF*) the method returned true, meaning the conversion was successful.

When running the tests on the smaller files created by us (which cover conversions in both directions), the method returned true in both cases.
Validating the RDF* to PG, PG to RDF* and SPARQL* to SPARQL conversions are currently not possible, this is discussed more in section 6.1.3.
6. Discussion

In this section the results and the method will be discussed and analysed. We will try to bring clarity to why the results turned out the way they did and how the method could be improved.

6.1 Results

The discussion of the result has been divided into three subchapters, the same as in chapter 5.

6.1.1 Conversion times

The measurements in Graphs 3-6 clearly show a linear pattern between the file size and the conversion time for all conversions. This makes sense considering the contents of the datasets used as File 1 and File 2 only contain reifications/nestings. This means all the data will be processed in the same way. If, however the datasets were a mix of regular triples and reifications/nestings a different curve might emerge. Unfortunately, mixed data sets are not as common and the ones that exist do not keep track of mix ratio, which is why such data sets were not used. In the context of PG conversions, the content shouldn’t have a big impact on the speed as all data is processed in a similar manner anyways.

When comparing the RDF* to RDF with the RDF to RDF* conversion times we expected similar conversion rates as both perform two readings of the file and have a similar program structure. However, looking at Table 2 and comparing the slopes it appears that RDF* to RDF is significantly slower which is surprising. Upon investigating a possible reason for this outcome, we discovered what we believe is the cause. In a Turtle* file the data is much more compressed than in a Turtle file resulting in a smaller file (since each nesting is equivalent to one reification statement). This theory is strengthened by Graph 1 and Graph 2 which shows similar conversion times for similar amount of metadata triples. For instance, our File 1 and File 2 differs a lot in size but actually contains roughly the same amount of data which has to be processed by the programs in a similar way. Therefore, Table 2 is somewhat misleading as it shows the relationship between conversion time and file size. It would be more fair to examine the slopes produced from comparing the relationship between conversion time and amount of nestings/reifications and regular triples. This logic also explains why the RDF to RDF* is faster than the RDF* to PG conversion even though the latter performs one less reading.

When comparing RDF* to PG with PG to RDF* we expected fairly similar conversion times. However, as seen in Table 2 the conversion rate for PG to RDF* was around three times as quick. This is in spite of both programs only performing one reading of the input file(s). We believe one of the contributing reasons for this is that the RDF* to PG conversion performs a substantial number of lookups in various containers, in contrast to the reversed conversion.
which hardly does any lookups. Another factor could once again be that the data is more compressed in one format than the other, though this would have to be investigated further.

If we look at the RDF* to RDF conversion contra the RDF* to PG conversion which uses the same dataset, the latter conversion is almost twice as fast than the former according to Table 2. This is congruent with the fact that the RDF* to RDF makes two readings of the input file and very few actions are made in the first lap.

6.1.2 Memory usage
Graph 9 and Graph 10 shows a linear pattern in the memory usage in relation to the file size. Both these conversions save almost all triples in the memory due to the content of the files containing only reifications/nestings. Since the number of reifications/nestings increases linearly to the file size as shown in Graph 7 and Graph 8 its only reasonable that the memory usage increase linearly with the file size as well.

Graph 12 also shows an expected outcome since the RDF* to PG conversion is always going to store most of the input data, regardless of the content of the input file.

Since the PG to RDF* barely stores any information in the memory the results shown in Graph 11 is exactly what we expected, a horizontal line.

An important discovery is the fact that the memory usage exceeds the size of the file in the tests displayed in Graph 9, Graph 10 and Graph 12. It does however have to be acknowledged that the type of content in the file plays a crucial role in the memory usage for the RDF* to RDF and RDF to RDF* conversions as seen in Table 3 and Table 4.

Looking at Table 5 we once again see results which might be odd as for example the RDF to RDF* conversion has a much higher memory usage efficiency than the RDF* to RDF. As previously mentioned the content matters. Looking at Graph 7 and Graph 8 it is shown that the both programs saves approximately the same number of triples in the memory even though the file size differs a lot. Since Table 5 shows the link between memory usage and file size the memory usage efficiency can be misleading.

One last thing worth mentioning is that for very large (roughly 2 GB or larger) RDF* and RDF files which consists of only nestings / reifications the programs will overflow the memory of a pc with 8 GB ram. This type of larger files would have to be split into smaller files, alternatively they would have to be run on a computer with more ram.

6.1.3 Validation of conversions
The isomorphic test on File 1 (RDF to RDF* and back to RDF) came back false due to the reason explained in section 5.4. It is sadly not uncommon that real world data sets are
flawed and contain errors of different kinds. However, there might be other factors (besides those illegal statements) that can cause the method to return false that we are unaware of. Yet, our suspicion is that isomorphic conversions are successful, as the tests on the small error free files also came back true.

When testing File 2 (RDF* to RDF and back to RDF*), the test returned true, meaning isomorphism was achieved when comparing the original dataset with the resulting file received after conversions.

When it comes to converting RDF* to PG and back to RDF*, or PG to RDF* and back to PG, there is no point in performing any kind of isomorphic test due to the fact that the before and after file cannot ever be isomorphic. This is because the Amazon Neptune format is expected as input when converting to RDF*, but Hartig’s format is used for output when converting to PG because it’s lossless (see section 2.4 and 4.2.1). The reason for not implementing support for Hartig’s format as input (when converting from PG to RDF*) is simply because it’s a new format with no data sets currently using it.

As a consequence of this manual controls are made instead when converting from and to PG, which is less than optimal. This makes it extremely hard to perform sufficient testing to cover all scenarios. It introduces the possibility of human error as well. This also applies to the SPARQL* conversions which suffers from the same problems. A more desirable approach would have been to create a tool that verifies the conversions but developing such a tool would be too time consuming for this thesis.

6.2 Method

We will evaluate our method from these three aspects: replicability, reliability and validity. From a replicability standpoint we believe this thesis can be reproduced as all essential frameworks used are open source e.g. Jena, RDFstarTools, Apache Commons. RDF and PG datasets are publicly available. However, RDF* datasets aren’t as widespread yet and might be more difficult to find.

Regarding reliability, similar results should be achievable but factors such as content of the datasets, computer specifications, and code structure can all have an impact on the results.

The validity of our method is perhaps the weakest part of this thesis. There are two major issues.

1. There is currently no way to compare two graphs expressed in different formats.
2. The subject of a reification statement is never saved when converting from RDF to RDF*, and in the reverse conversion it receives a newly generated id.
The first point severely limits our capability to confirm that the converted graph is an identical representation of the original graph.

The second point means that converting from RDF to RDF* and back will never result in a file that is identical to the original. Thus, comparing if they are equivalent is futile. Therefore, the `isIsomorphicWith` method was used, since it ignores the label on blank nodes when comparing graphs.

Another part that could be improved is regarding the datasets. For one, more datasets could be used where the content is a mixture of regular triples and reifications/nestings. Additionally, datasets of larger size would also contribute to a more reliable result.

6.2.1 Source criticism
There is a fair number of sources used. However, more scientific sources could have been cited. The RDF* format is relatively new which limits the amount of available scientific sources in the field. The scientific sources that we did use have all been cited by others, in particular Hartig’s work which this thesis heavily relies on. Most of the non-scientific sources do, however, come from the W3C organization which is the official managing organization of the RDF standards. Some of the sources have ties to Neo4j which makes them potentially biased in the use and effectiveness of property graphs but we mainly use these sources for broader statements.
7 Conclusion

Our goal with this thesis was to allow for the use of RDF* and thereby bridging the gap between RDF and PG. To achieve this our research had to answer the following questions:

Is it possible to develop a program in java that can convert in a streaming fashion between arbitrarily large datasets without loss of data from a RDF* file serialized in Turtle* to a RDF file serialized in Turtle and vice versa?

The short answer is yes. When performing one-way conversion starting from either format, it is lossless and streamable. However, if starting from RDF and converting to RDF* and back we can’t guarantee with 100% certainty that the graph is the same. Therefore RDF* is possible whereas RDF to RDF* is most likely possible.

Is it possible to develop a program in java that can convert in a streaming fashion between arbitrarily large datasets without loss of data from a RDF* file serialized in Turtle* to a PG file serialized in CSV and vice versa?

Strictly speaking, if the PG file follows the Amazon Neptune format [13] in a PG to RDF* conversion, there can be some minor data distortion when it comes to “Date”-types as explained in section 4.3. Also, there is no proper validation on these conversions to verify the integrity of the data. Thus, the answer to this question should be no, but since CSV can be serialized in different ways and with different rules it is perhaps more appropriate to say the answer to this question is undecided.

Assuming the conversions are possible to implement, what kind of performance can be expected for each type of conversion on datasets of varying sizes in terms of conversion speed and memory usage?

In terms of conversion rate:

- RDF to RDF* takes approximately 0.0873 seconds to convert one MB data
- RDF* to RDF takes approximately 0.1566 seconds to convert one MB data
- PG* to RDF* takes approximately 0.0292 seconds to convert one MB data
- RDF* to PG takes approximately 0.0907 seconds to convert one MB data

However, as we mentioned previously, the content of the file will affect the conversion rate.

In terms of memory usage efficiency

- RDF to RDF* stores approximately 1.3891 MB in memory per MB read
- RDF* to RDF stores approximately 3.8541 MB in memory per MB read
- PG* to RDF* stores approximately 0 MB in memory per MB read
- RDF* to PG stores approximately 3.0388 MB in memory per MB read

In this case, the content of the file also heavily affects the outcome.

Is it possible to develop a program in java that can convert from a SPARQL* query to an equivalent SPARQL query?
The program developed seems to be able perform this conversion. However, as there is no database management system which can execute SPARQL* queries, we can’t confirm that a SPARQL* query would return the same data as its equivalent SPARQL query. Therefore, the conclusion is that it is possible. However, we can’t confirm that our developed program lives up to the requirements.

For future work it would be necessary to develop proper testing tools to validate all conversions. It would also be helpful to develop a RDF* conversion tool which can output different PG formats. An even better alternative would be to define an official standard PG format.
8 References


