Efficient update of OTTR-constructed triplestores

Update algorithms for the OTTR framework

Magnus Wiik Eckhoff, Preben Zahl

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2023

Efficient update of OTTR-constructed triplestores

Update algorithms for the OTTR framework

Magnus Wiik Eckhoff, Preben Zahl

Efficient update of OTTR-constructed triplestores

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Reasonable Ontology Templates (OTTR) is a language designed to improve the efficiency and quality of building, using, and maintaining knowledge bases. OTTR introduces templates to create patterns of RDF triples. Templates are instantiated by template instances, which can be expanded to an RDF graph, and then stored in a triplestore. It is desirable to use the template instances to dictate the content of the triplestore. Currently, OTTR has no way of updating the triplestore when a change to the template instances occurs, besides rebuilding all triples. In this thesis, we have created several algorithms to more efficiently update the triplestore based on changes to template instances. Our algorithms aim to update only the parts affected by the change. First, we created a simple solution with excellent performance, but strict assumptions about input data. Then, we created other solutions that remove one or several assumptions. All solutions significantly outperform OTTR's current solution in a typical use case. We investigate the performance of the different solutions and compare them to each other. The result of this thesis lays the groundwork for how updates can be part of OTTR in the future.

Acknowledgement

We thank our supervisor, Leif Harald Karlsen, for investing a lot of time in guiding us through this thesis. He has given excellent advice and feedback on our thoughts, which made this project more interesting and enjoyable. We also want to thank Ane Moen Mahlum and Julie Akselberg for their support and for listening to excitement about distant topics. Thanks to our families for their interest in and support of our studies. Additionally, we want to thank Christian Mahesh Hansen and Martin G. Skjæveland for their valuable input. Finally, thanks to our friends for their discussions and encouragement.

Contents

List of Figures

Listings

Part I

Introduction and background

Chapter 1

Introduction

With the ever-increasing importance of information, data has quickly emerged as a highly valuable resource. Being able to quickly access, link, and interpret meaningful data is more important than ever. With this in mind, the semantic web was created as an extension of the World Wide Web that enables a web of linked data through defined standards and technologies [45].

It is vital to make the web of data not only a collection of datasets, but a standard format for data and their relationships among each other. World Wide Web Consortium (W3C) [1], the organization responsible for defining these standards, has created a common data format for this purpose called Resource description framework (RDF) [9]. Additionally, W3C has provided standards for accessing and reasoning over the data. This is made possible through various technologies such as OWL, SPARQL, and SHACL [22]. RDF enables connecting data from different systems by expressing everything as triples. Triples connect two data resources together with a relationship. A large collection of RDF triples are referred to as an RDF graph. RDF graphs are typically stored in specialized databases called triplestores [44].

Big RDF graphs can be built by creating many RDF triples of linked data. The process of creating these graphs can be time-consuming as the number of triples can quickly become high. Moreover, patterns in the data have to be manually created, causing repetition and the possibility of errors. Reasonable Ontology Templates (OTTR) is a framework for making ontologies and RDF graphs that address these problems. This is done by introducing templates as a way of creating parameterized modeling patterns [46]. An instance of a template can be specified with appropriate arguments and expanded into an RDF graph.

Using OTTR to create and maintain large RDF graphs is a significant improvement compared to pure RDF. However, whenever the OTTR instances or templates are changed, all OTTR instances must be expanded

in order to rebuild the graph. This can be time-consuming, especially when the graph is large. Therefore, it is desirable to be able to efficiently update the RDF graph once a change occurs.

Let's say you are a company creating large ships. For this purpose, a large ontology is created in OTTR, modeling all ship parts, ranging from bolts and nuts to water pipes. The parts are related to each other through different properties based on their compatibility. For example, if a nut fits on a bolt, a triple describing this fits-relation is created. The ontology is stored in a triplestore, which makes it possible to modify and query the data. Periodically new information is added to the triplestore when new parts are introduced. Information is changed or removed when parts are changed or no longer in use. For an ontology expert working with a triplestore, it is desirable to process changes quickly, as this makes for a better developing experience that is less prone to errors [47]. Engineers planning ship construction use the triplestore to validate the correctness of their plans. For them, the triplestore must be up-to-date and available. In the case where the ontology is very large, a single rebuild might take hours. During this time, no validation can be performed, making the ship construction planners less efficient. If the update took minutes instead of hours, one can expect productivity for both the ontology expert and the ship planning engineer to increase.

In this thesis, we will investigate whether it is possible to create an efficient update algorithm for OTTR. Secondly, we will explore how the input data affects the algorithm's performance. Lastly, we will evaluate whether the algorithm is a good addition to OTTR.

1.1 Outline

This thesis is outlined as follows:

- **Chapter 2: Background**
  This chapter covers the background information relevant to this thesis. We provide an overview of the Resource description framework (RDF). Following this, we introduce SPARQL, a query language over RDF. Next, we cover a common extension to RDF and SPARQL; RDF-star and SPARQL-star. Furthermore, we introduce OTTR and its characteristics. Lastly, we present difference algorithms and how they are used.
- **Chapter 3: Problem specification**
  In this chapter, we present the problem specification in more detail and an example of how an algorithm should work. The scope of the thesis is presented together with a description of how we will evaluate the results.

- **Chapter 4: Current solution: Rebuild solution**
  This chapter describes how updates in OTTR are currently performed.

- **Chapter 5: Solution assessment**
  Firstly, this chapter describes the typical use case for OTTR users. Secondly, we describe what we test and how we test the solutions in this thesis.

- **Chapter 6: Simple solution**
  This chapter outlines a *Simple solution*. It describes how the solution works and two assumptions being made: the *duplicate assumption* and the *local blank node assumption*. Following this, it presents and discusses its performance based on results from testing.

- **Chapter 7: Implementation**
  In this chapter, the general implementation of the testing program is discussed. Following this, it describes the implementation of the *Simple solution*.

- **Chapter 8: Removing the duplicate assumption**
  The *Duplicate solution* is presented. It describes how the duplicate assumption can be removed. An outline of the solution is presented together with details of the implementation. The results from testing are presented and discussed.

- **Chapter 9: Removing the local blank node assumption**
  The *Blank node solution* is presented and discussed. This chapter outlines how to remove the local blank node assumption and provides details on the implementation. The *Blank node solution* is tested and compared to rebuilding.

- **Chapter 10: Combined solution**
  This chapter explains how the *Duplicate solution* can be combined with the *Blank node solution* to create a working update algorithm with few limiting assumptions.

- **Chapter 11: Comparison**
  In this chapter, we compare the different solutions to each other by doing experimental evaluation.

- **Chapter 12: Discussion**
  In this chapter, we discuss the findings of this thesis. We evaluate if we managed to create a suitable update algorithm for OTTR and discuss if it is suitable to become a part of OTTR. Then, we look at the limitations of our evaluations.

- **Chapter 13: Related Work**
  This chapter examines existing works related to this thesis and evaluates their relevance to our work.

- **Chapter 14: Conclusion**
  In the final chapter, we summarize the main findings of this thesis. Lastly, we provide suggestions for future work.

Chapter 2

Background

OTTR is the framework of focus in this thesis that we want to extend with and update algorithm. OTTR is created as an extension of semantic technology to better interact with RDF graphs. In this chapter, we will introduce the parts of semantic technology relevant to this thesis.

In Section 2.1, we look at RDF, a standard for modeling and exchanging data on the Web. This section includes the central parts of the syntax as well as its properties. Following this, Section 2.2 looks at SPARQL, a query language over RDF. Moreover, in Section 2.3, we examine RDF-star, an extension to RDF, and SPARQL-star, its corresponding query language. Section 2.4 covers OTTR, a language used to create and represent RDF graphs. Lastly, Section 2.5 covers briefly what a difference algorithm is and the GNU Diffutils interface.

2.1 RDF

Resource description framework (RDF) is a standard for describing data and is an essential element in the vision of a semantic web. This section provides a brief description of RDF based on the W3C specification for RDF 1.1 [9].

RDF allows us to unambiguously identify resources through unique identifiers called IRIs, making connecting data from several places across different systems very simple. The main idea of RDF is that everything can be expressed as triples. A triple is a sentence consisting of three resources: a subject, a predicate, and an object in that order. For example "`(Ola likes food)`". Here, "`Ola`" is the subject, "`likes`" is the predicate, and "`food`" is the object. The end of a triple is marked by ".". A Triple is also referred to as an *RDF statement*, or just *statement*. Resources in RDF are IRIs, blank nodes, or literals, which we will explain in the following sections. A collection of

these triples is called an RDF graph. RDF graphs are commonly stored in purpose-built databases called triplestores [44].

IRI

An IRI (Internationalized Resource Identifier) is a globally unique string that identifies a resource. In RDF, almost every resource has an associated IRI to identify it. The exception is data values and blank nodes, which we will discuss in the following sections. By using IRIs instead of local IDs, we can combine our graph with another graph that uses the same IRIs for the same resources, without the need to match the data explicitly. Using existing IRIs is much better than creating new ones, as it is easier to connect to existing datasets. URLs, emails, and ISBNs are examples of IRIs. In RDF, IRIs are surrounded by angle brackets, like this "<IRI>".

```
1 <https://en.wikipedia.org/wiki/John_Adams>
2 <http://xmlns.com/foaf/0.1/knows>
3 <https://en.wikipedia.org/wiki/George_Washington> .
```

Listing 2.1: IRI example

The example in Listing 2.1 illustrates a triple with three IRIs, that all are unique and unambiguous. If we go to the page https://en.wikipedia.org/wiki/George_Washington, we will see the first American president and not another person also named George Washington. The predicate in this triple is an URL to a page that describes the knows relation. In this way, we can precisely state that John knows George.

Prefixes and Qnames

In the example in Listing 2.1, there is a triple with three entire IRIs. This way of writing triples is expressive, but inconvenient for humans because IRIs are long and hard to read. Therefore, we usually use Qnames [10] to make them more readable. Qnames consist of a prefix and a local part. Websites often define multiple concepts within the same domain, resulting in the beginning of the IRIs being the same for multiple resources, only differentiated by the ending. We define prefixes for the beginnings and add local parts for the endings.

```
1 @prefix foaf: <http://xmlns.com/foaf/0.1/>
2 @prefix wiki: <https://en.wikipedia.org/wiki/>
3
4 wiki:John_Adams foaf:knows wiki:George_Washington .
```

Listing 2.2: Qname example

In the example in Listing 2.2, we use Qnames three times; `wiki:John_Adams`, `wiki:George_Washington` and `foaf:knows`. This triple is equivalent to the one in Listing 2.1.

Literals and Blank nodes

In some cases, representing data without giving it an IRI is desirable. It would not be beneficial to link all occurrences of the number "1" together due to how common it is. To achieve this, literals are used. Literals are used to represent data values and their associated type. By default, all literals are interpreted as a string, but a datatype can be specified by appending the value with a data type. For example, `"12^^xsd:int"` gives the value `12` the `xsd:int` datatype, which is a standard integer [5].

Blank nodes behave like resources without an IRI. They enable the specification of resources we cannot or do not want to give an IRI. Blank nodes can be used when linking multiple resources, like an address consisting of a street name, street number, and city. Some RDF syntaxes have blank node identifiers available in the local scope. Blank nodes are represented by `"_:name"`, where "name" is the name of the blank node. Blank nodes can also be represented by `"[]"`, but we will use `"_:name"` in this thesis. Listing 2.3 is an example where someone we call "X" is 20 years old and knows John.

```
1 @prefix foaf: <http://xmlns.com/foaf/0.1/>
2 @prefix wiki: <https://en.wikipedia.org/wiki/>
3 @prefix xsd: <http://www.w3.org/2001/XMLSchema#>
4
5 _:X foaf:knows wiki:John_Adams .
6 _:X foaf:age 20^^xsd:Integer .
```

Listing 2.3: Literals and blank nodes

Format

There is a variety of syntax notations for writing RDF triples. All previously shown examples are in turtle format [6]. Turtle is a widely used format, as it is easy for humans to read. We will, therefore, continue to use the turtle notation when showing examples of RDF. In addition to the already shown features, turtle has syntactic sugar for a single subject used in several triples. This is done by adding `";"` instead of `"."` after a triple, then continuing with the predicate and object of a new triple. Similarly, `","` can be used to reuse the same subject+predicate in multiple triples. These features are common and will be used in RDF examples. This syntactic

sugar is shown in Listing 2.4, where someone (blank node) knows John and George, and is interested in freedom.

```
1 @prefix foaf: <http://xmlns.com/foaf/0.1/>
2 @prefix wiki: <https://en.wikipedia.org/wiki/>
3 @prefix xsd: <http://www.w3.org/2001/XMLSchema#>
4
5 _:X foaf:knows wiki:John_Adams ,
6                 wiki:George_Washington ;
7    foaf:interest wiki:freedom .
```

Listing 2.4: Turtle format

Properties of RDF

In RDF, there is an IRI for almost everything, even relations. In the earlier example (`Ola likes food`), we used the relation `"likes"` as a predicate. However, this is also a resource that can be described. For example, (`likes a predicate`) where `"likes"` is the subject and is described as a predicate. This way, data can specify other data's semantics, but RDF also allows it to specify its own semantics. The statement: (`owl:sameAs owl:sameAs owl:sameAs`) is a self-describing statement where `owl:sameAs` describes itself. The ability to describe the relations in an RDF graph differentiates it from graphs in graph theory, where giving edges attributes is impossible.

RDF's simple structure allows it to express any statement. There are no data type requirements or conflicting statements that restrict a valid statement, and this makes RDF suitable for data integration. It is always possible to add another statement to an already existing RDF graph or combine two different graphs by simply adding all statements together.

Vocabularies

To make the RDF graph use as few new IRIs as possible, we look to common vocabularies that already define many resources. In this way, our graph can be combined with other graphs that use the same IRIs for the same resources. An example of this is the "Friend of a friend" (foaf) vocabulary. Foaf is devoted to linking people on the web and has defined many "Social network" resources like the relation `knows`.

RDF and RDFS [21] are examples of other famous vocabularies. RDF contains the property `type`, but this is usually abbreviated as "a" instead of `rdf:type`. This is used to specify the type of a resource. These vocabularies also introduce several other data-modeling terms like classes, properties,

resources, subclasses, sub properties, domain and range.

2.2 SPARQL

SPARQL is a query language over RDF [4]. It is used to create, manipulate and extract data from triplestores, similar to the query language for relational databases SQL [19]. This section provides an overview of SPARQL fundamentals and a selection of operations relevant to this thesis.

Basics

A SPARQL query consists of multiple optional and non-optional parts. The first part of a query is the prefix declarations. This is similar to how RDF prefixes can be shortened using Qnames as discussed in Section 2.1. The prefix `foaf` is defined on line 1 in Listing 2.5.

The second part is where we specify what should be done and returned. There are different kinds of clauses, for different operations. For example, select, construct, delete, insert, and more. In the `SELECT` clause, we choose which data to be the result of the query. We define variables which we further describe in a corresponding `WHERE` clause. Line 2 in Listing 2.5 specifies that we want to `SELECT` one variable: the `?name`. `SELECT` returns all or a subset of variables bound in the `WHERE` clause. A `CONSTRUCT` query can be used as an alternative to `SELECT`. It uses a `WHERE` clause in the same way, but creates a graph to return instead of just values [26].

Following this part is the dataset definition. As a dataset can consist of multiple sub-graphs, SPARQL provides the keyword `FROM` to specify which sub-graph we query over [56]. The query will be evaluated on the default graph if a `FROM` clause is not specified.

The `WHERE` clause is used to specify a graph pattern to search for. This pattern should include the variables specified in the `SELECT` clause, but it typically also includes other variables, IRIs, or literals [4]. In the example shown in Listing 2.5 on lines 5 and 6, we are looking for people who are 67 years old (line 5) and their names (line 6). Upon execution of the query, the simple graph pattern is matched with a subgraph of the RDF data. In SPARQL, joins occur as parts of the triple patterns. If the same variable name is used in several triples, they are joined, combining their information.

Lastly, we have the query modifiers as shown on line 8. These are operations that modify or change the result of the query. The `LIMIT` operator limits the number of results returned, 1 in this case. There are many other operators, for example, `ORDER BY`, `GROUP BY`, and `OFFSET`, that

order, group, and skip results, respectively.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
FROM <http://norway.no/population>
WHERE {
  ?person foaf:age "67" .
  ?person foaf:name ?name .
}
LIMIT 1
```

Listing 2.5: SPARQL query

```
:id1 foaf:name "William" .
:id1 foaf:age "67" .
:id2 foaf:name "George" .
:id2 foaf:age "42" .
:id3 foaf:name "Thomas" .
:id3 foaf:age "67" .
```

Listing 2.6: RDF graph being queried

name
Thomas

Table 2.1: Result of the query

Performing the query in Listing 2.5 over the RDF graph in Listing 2.6 Results in Table 2.1. Note that while both "William" and "Thomas" match, the `LIMIT 1` operator limits the result to only one of the matches.

Update

We will now briefly outline the W3C recommendation "SPARQL 1.1 Update" [40]. SPARQL offers two operations to manipulate graphs, `INSERT` and `DELETE`. The operations change the existing graph in a triplestore by inserting or deleting a triple, respectively. An operation can be made conditional by using the `WHERE` clause. In Listing 2.7, we insert a triple stating that every person of age 67 is a senior.

```
1 INSERT {
2     ?person ex:ageGroup :senior .
3 }
4 WHERE {
5     ?person foaf:age "67" .
6 }
```

Listing 2.7: INSERT query

Updating a triple in the graph can be viewed as deleting the old triple and inserting a new one with the updated value. This is a common usage pattern; therefore, SPARQL includes a `DELETE`/`INSERT` operation. The `DELETE`/`INSERT` operation is interpreted as one operation rather than a sequence of operations. This results in a more powerful query than a sequential one, as the `WHERE` query will match the triples that will be updated. In the Listing 2.8, the name is changed from Bill to William for every Bill that is 30 years old.

```
1 DELETE { ?person foaf:name 'Bill' }
2 INSERT { ?person foaf:name 'William' }
3 WHERE {
4         ?person foaf:name 'Bill' .
5         ?person foaf:age "30" .
6 }
```

Listing 2.8: DELETE/INSERT query

Advanced operations

Apart from the previously mentioned operations, SPARQL provides a wide range of additional functionality. We will now explore a selection of SPARQL functionality relevant to this master thesis.

The example in Listing 2.9 shows a query that selects the name of everyone with an address that no one else has. In other words, the name of all those who live alone. This is achieved by combining a sub-query, an aggregator, and a filter. The sub-query from lines 6 to 11 is the first part of the query to be evaluated. On line 7, we see an example of an aggregation operator: `count`. This operator counts the number of occurrences of the `?address` variable bound in the subsequent where clause. The result of this count is bound to a variable `?num_at_address` with the AS keyword.

A filter is applied to the result of this sub-query. By specifying `?num_at_address = 1`, only the cases where `?address` is bound precisely one time in the subquery are returned. Finally, the graph in the where clause is combined with the graph in the subquery and is evaluated over

the triplestore.

```
SELECT ?name
WHERE {
    ?person foaf:name ?name .
    ?person foaf:address ?address .
    # subquery
    {
        SELECT (count(?address) AS ?num_at_address)
        WHERE {
            ?person foaf:address ?address .
        }
    }
    FILTER (?num_at_address = 1)
}
```

Listing 2.9: Subquery with aggregate function

When we wish to perform the same action for multiple elements, the `VALUES` clause allows us to specify a set of possible values for one or more variables. In Listing 2.10, we specify that the `?name` variable can have the value of either "William" or "Bill".

```
SELECT ?name ?age
WHERE {
    ?person foaf:name ?name .
    ?person foaf:age ?age .
    VALUES (?name) {
        ("William")
        ("Bill")
    }
}
```

Listing 2.10: Query with values. Name must be "William" or "Bill"

When only parts of the result is aggregated, as shown in Listing 2.11, it is necessary to specify how to combine the aggregated variables with the other variables. We can do this by grouping the results by a variable. An alternative approach to using the `VALUES` clause when assigning multiple values to a variable is by using `FILTER (?variable IN ( element1 element2 ...))`.

```
SELECT ?name (COUNT(?known) AS ?nr_friends)
WHERE {
    ?person foaf:name ?name .
    ?person foaf:knows ?known .
    FILTER (?name IN ( "William" "Bill" ))
}
GROUP BY ?name
```

Listing 2.11: Query with group by. Find the number of friends called "William" or "Bill"

2.3 RDF-star and SPARQL-star

RDF-star, often referred to as RDF*, is an extension of RDF that allows statements to be represented as a single resource [31]. This enables the creation of statements about statements. Being able to describe the statements in a graph separates RDF* from standard RDF, which is only intended to describe the resources in a graph [55]. However, RDF reification makes this possible, by having separate triples to specify subject, predicate, and object to a resource, then this resource can be referenced to as the triple. While possible, this is unpractical to use, as it requires describing every node in a triple with its own triple [42]. We will now look at the idea behind RDF* and how SPARQL* enables querying over it.

Let us say that we have a small graph with a single triple (`<Ole> :knows <Kari>`). With RDF, we have no simple way of describing when this relation started. We could create another triple with a :gotKnown relation, but it is not explicit that those two relations are related. That requires implicit knowledge about the relations. With RDF*, however, we can describe the statement (`<Ole> :knows <Kari>`). We can treat the statement as a single resource by surrounding it with double angle brackets, `<< statement >>`. In Listing 2.12, we have added a statement about a statement. We now know when the :knows relation between `<Ole>` and `<Kari>` started.

```
<Ole> :knows <Kari> .
<< <Ole> :knows <Kari> >> :started
   "2018-08-28"^^xsd:date .
```

Listing 2.12: RDF* example

SPARQL-star, or SPARQL*, is an extension to SPARQL that allows querying over RDF* graphs. The syntax with angle brackets can also be used here to represent statements as resources. SPARQL* also introduces some functions [7] that we do not use in this thesis. In Listing 2.13, we ask for all triples with a date newer than "2000-01-01". This would return (`<Ole>`

`:knows <Kari>`) from Listing 2.12.

```
1 SELECT ?s ?p ?o
2 WHERE {
3     <<?s ?p ?o>> :date ?d
4     FILTER (?d < "2000-01-01"^^xsd:date)
5 }
```

Listing 2.13: SPARQL* example

RDF* and SPARQL* have become quite popular and are enabled by default in Fuseki [11], our triplestore of choice. W3C and other companies are working on making RDF* and SPARQL* the standard for representing metadata with triples [55].

2.4 OTTR

Reasonable Ontology Templates (OTTR) is a language with tools for representing and creating RDF graphs. OTTR has templates to represent RDF patterns and creates a higher level of abstraction by using these as building blocks rather than RDF triples, allowing users to easily build and maintain larger ontologies. We will now present the basic building blocks of OTTR as well as some of its characteristics.

Templates

An OTTR template, or just template, consists of a signature and a body. The signature on line 1 in Listing 2.14 specifies the template name and the parameters, similar to a function in other programming languages. The body, written on lines 2 and 3, contains instances of other templates.

```
1 ex:Person[?person, ?name, ?age] :: {
2     ottr:Triple(?person, foaf:name, ?name),
3     ottr:Triple(?person, foaf:age, ?age)
4 }
```

Listing 2.14: OTTR template

In Listing 2.14, a person template takes `?person`, `?name`, and `?age` as parameters. The body contains two instances of the `ottr:Triple` template, one with the person's name and one with age. `ottr:Triple` is a base template. Base templates are special templates that do not contain a body. They usually represent a fundamental data structure in the underlying

technology. OTTR currently contains only one base template, ottr:Triple. This base template represents an RDF triple and takes three arguments; subject, predicate, and object. It is also possible to define custom base templates if needed. Each instance of the `ex:Person` template will create two `ottr:Triple` instances, which we can convert to RDF triples with the given arguments.

Instances

OTTR can represent an RDF graph by creating OTTR template instances of the templates. Template instances, OTTR instances, or just instances all refer to the same unless specified otherwise. In Listing 2.15, there is an example of an instance of the ex:Person template.

```
1 ex:Person(<bob>, "Bob Marley", 36) .
```

Listing 2.15: OTTR instance

The expansion of an OTTR instance is done by populating the instances in the template body with the corresponding arguments, and then continuing with expanding them again until all instances are base templates, which cannot be expanded [32]. The instance of the `ex:Person` template (Listing 2.15) expands to the instances in Listing 2.16. These are instances of `ottr:Triple`, which is a base template that will not expand to anything else. `ottr:Triple` represents RDF triples, so these instances can be converted to the corresponding RDF triples as shown in Listing 2.17.

```
1 ottr:Triple(<bob>, foaf:name, "Bob Marley") .
2 ottr:Triple(<bob>, foaf:age, 36) .
```

Listing 2.16: OTTR expansion

```
1 <bob> foaf:name "Bob Marley" .
2 <bob> foaf:age 36 .
```

Listing 2.17: RDF representation

We now have a more efficient way of creating RDF graphs. The efficiency is especially prevalent when the same pattern is created many times and when a single template results in many triples.

Nested templates

A template contains instances of other templates, with the exception of base templates; this makes templates a recursive data structure. By nesting

templates, we can represent complex structures, creating several layers of abstraction. Listing 2.18 is an example of a template for a couple that takes two persons as arguments and creates one ex:Person instance for each. Then, the ex:Person instances would further expand to `ottr:Triples` as previously shown.

```
ex:Couple[?p1, ?p1Name, ?p1Age, ?p2, ?p2Name, ?p2Age]
   :: {
    ex:Person(?p1, ?p1Name, ?p1Age),
    ex:Person(?p2, ?p2Name, ?p2Age)
}

ex:Person[?person, ?name, ?age] :: {
    ottr:Triple(?person, foaf:name, ?name),
    ottr:Triple(?person, foaf:age, ?age)
}
```

Listing 2.18: Nested OTTR templates

Types

One key benefit of OTTR is the type system. OTTR allows you to specify the type of the parameters in the templates, and will catch errors early if it receives data in the wrong format. In this way, OTTR can contribute to a robust and understandable system. While the type system is an essential part of OTTR, it is not important for understanding the topics discussed in this thesis. We will now show a simple example of how types in OTTR are specified. For more information on OTTR types, see [34].

Specifying a type in OTTR is done by writing the type before an argument. This is done in Listing 2.19, where the person must be an IRI, the name a string, and the age an integer. OTTR has several standard types known from RDF and XSD, but also some OTTR-specific types, like *ottr:IRI*, *none*, and list types. OTTR also checks for type consistency, which means a parameter cannot be forced to have multiple incompatible types.

```
ex:Person[ottr:IRI ?person, xsd:string ?name,
   xsd:integer ?age] :: {
    ottr:Triple(?person, foaf:name, ?name),
    ottr:Triple(?person, foaf:age, ?age)
}
```

Listing 2.19: OTTR template with types

Other OTTR constructs

OTTR supports optional arguments [34]. A parameter can be marked optional with a question mark "?". `ottr:none` is used to represent a missing or no value. If "none" is used as an argument to an optional parameter, then the instance is expanded normally as for any other value. If "none" is used as an argument to a non-optional parameter, then the whole instance is not expanded. Arguments can also have a default value, marked by setting `?parameter=value`. The default value will be used when the argument is "none".

Arguments can also be lists of a specified type. Parameters can be marked as a list with `List<type>`, where type is the type of the elements. If we want to create several instances with the list-elements as arguments rather than passing the entire list to a single instance, we must specify an expansion mode and mark the parameters with ++. There are currently three expansion modes: `cross`, `zipMin`, `zipMax`. If there is only one list argument, then it does not matter which expansion mode we choose. An expansion of a single list is shown in Listing 2.20. In this example, Bob would end up knowing both Lisa and Roger, and they would know him as seen in Listing 2.21. Here, cross was used, but `zipMin` and `zipMax` would have given the same result.

```
1  ex:Friends[?person, List<ottr:IRI> ?friends] :: {
2      cross | ex:Friend(?person, ++?friends)
3  }
4
5  ex:Friend[?person1, ?person2] :: {
6      ottr:Triple(?person1, foaf:knows, ?person2),
7      ottr:Triple(?person2, foaf:knows, ?person1)
8  }
9
10 ex:Friends(<bob>, (<lisa>, <roger>))
```

Listing 2.20: OTTR, one list parameter

```
1 ex:Friend(<bob>, <lisa>) .
2 ex:Friend(<bob>, <roger>) .
```

Listing 2.21: One step of expansion of Listing 2.20

The different expansion modes behave differently when there are multiple list parameters. If the first parameter in ex:Friends also is a list, then we would need a way to specify which combination of list elements should be used as arguments together. `Cross` means using every combination, i.e., Cartesian product. `zipMin` traverses both lists simultaneously and stops when one list stops. `zipMax` is the same as `zipMin`, but continues until the longest list stops. When the shorter list stops, it continues with "none"

values instead. Examples of this can be seen in Listing 2.22 and Listing 2.23

```
ex:Friends[List<ottr:IRI> ?persons, List<ottr:IRI>
   ?friends] :: {
    expansion_mode | ex:Friend(++?person, ++?friends)
}

ex:Friend[?person1, ?person2] :: {
    ottr:Triple(?person1, foaf:knows, ?person2),
    ottr:Triple(?person2, foaf:knows, ?person1)
}

ex:Friends((<bob>), (<lisa>, <roger>))
```

Listing 2.22: OTTR, two list parameter

```
# expansion_mode = cross
ex:Friend(<bob>, <lisa>) .
ex:Friend(<bob>, <roger>) .

# expansion_mode = zipMin
ex:Friend(<bob>, <lisa>) .

# expansion_mode = zipMax
ex:Friend(<bob>, <lisa>) .
ex:Friend(ottr:none, <roger>) .
```

Listing 2.23: One step of expansion of Listing 2.22

Expansion

The expansion of instances continues until all instances are base templates and can be summarized in these points:

- If it is an instance of a base template, it doesn't expand as it has no pattern.
- If an argument has a list expander (++), the list is expanded, and new instances are created by applying a specified operation (cross, zipMin, zipMax). The expansion of this instance will be the collection of all the newly generated instances.
- If an argument has no value and its corresponding parameter is not optional and has no default value, then the result of the expansion is the empty set. If it has a default value, then that is used as the argument.

- Otherwise, create the new instances in the template body and replace the parameters with the corresponding arguments.

Lutra

Lutra is an open source implementation of the OTTR language [49]. It supports reading and writing OTTR templates and instances and expanding instances into RDF graphs. It is implemented in Java, and is available as a command line interface and as a library.

2.5 Difference algorithm

The first step in creating an efficient update algorithm is identifying what needs to be updated. In our case, this would be identifying the difference between the OTTR instance files before and after a change has occurred. Creating a representation of the difference between two strings can be done using a difference algorithm, commonly called a diff algorithm. There are multiple practical use cases utilizing diff algorithms, including version control systems like git [27] and data compression with delta encoding [36].

GNU Diffutils

GNU diffutils is an open-source package published under the GNU GPL version 3+ license [20]. The package contains a diff command which is used to show the difference between two files. The two files are compared on a line-by-line basis. The diff algorithm finds sequences of common lines together with groups of different lines called hunks [20]. The command minimizes the total hunk size and, in the process, maximizes the length of sequences of common lines.

Example

Given an instance file A

```
1 ex:Person(<111>, "Romeo") .
2 ex:Person(<222>, "Juliet") .
3 ex:Person(<333>, "Mercutio") .
```

And an instance file B

```
1 ex:Person(<111>, "Changed name") .
2 ex:Person(<222>, "Juliet") .
```

The output of a diff command between File A and File B

```
1 1c1
2 < ex:Person(<111>, "Romeo") .
3 ---
4 > ex:Person(<111>, "Changed name") .
5 3d2
6 < ex:Person(<333>, "Mercutio") .
```

The output of the diff command is a sequence of hunks. The hunks are first described by a change command as seen on lines 1 and 5 in the diff output, followed by the affected lines. A change command consists of 3 parts. Firstly is the interval of affected lines in the first file. Secondly is the operation to be performed. This is either change (c), delete (d), or insert (i). Lastly is the interval of lines affected in the second file.

Gnu Diff utils implement the Myers diff algorithm with a heuristic by Paul Eggert [38]. The algorithm is versatile and fits different types of input string pairs. It also scales with the size of the changes, meaning it performs well in cases where the number of changes is few compared to the total amount of data in the file. In addition, the Myers algorithm serves as the core of popular tools such as Google Diff match patch [28] and as the default algorithm in git diff [27].

Chapter 3

Problem specification

This chapter covers the objective of this thesis and a short description of the problem to investigate. Following this, is a short example of the intended behavior of an update algorithm. Further is a formal definition of the problem presented. Lastly, we cover the problem's scope and how this thesis' results will be evaluated.

In this thesis, our objective is to answer three questions:

- Is it possible to create an efficient update algorithm for OTTR
- How the input data affects the algorithm's performance
- Is the algorithm a good addition to OTTR

To answer if it is possible to create an efficient update algorithm for OTTR, we will investigate the following problem: Given a set of OTTR templates, a set of OTTR instances, and a triplestore consisting of the expansion of the instances over the templates. Suppose we now make a change to the instances. In that case, we want to create an efficient update algorithm that correctly updates the triplestore to reflect the changed instances. If the triplestore is equal to the expansion of the new instance file, we say that the instances and triplestore are *synchronized*. Conversely, if the triplestore is not equal to the expansion of the new instances file, we say that the instances and the triplestore are *desynchronized*. If we can create an algorithm that efficiently keeps the triplestore synchronized when a change is made to the instances, then it is possible to create an efficient update algorithm for OTTR.

To identify how the input data affects the algorithm's performance, we will measure the algorithm with different kinds of input. Since we cannot test all kinds of data, we will measure a selection that we expect to have the greatest impact on performance.

To answer whether the algorithm is a good addition to OTTR, we must identify the typical use case for an OTTR update algorithm and measure how our algorithm performs in such a case. If our algorithm gives correct results, outperforms OTTR's current solution, which is discussed in the next chapter, and would perform equally well if implemented in OTTR, then the algorithm would be a good addition.

Example

Given a template

```
ex:Person[?person, ?name, ?age] :: {
    ottr:Triple(?person, foaf:name, ?name),
    ottr:Triple(?person, foaf:age, ?age)
}
```

And a template instance

```
ex:Person(ex:Bob, "Bob", 32) .
```

This expands to the following RDF triples

```
ex:Bob foaf:name "Bob" .
ex:Bob foaf:age 32 .
```

If one were to change the name from "Bob" to "Bobby"

```
ex:Person(ex:Bob, "Bobby", 32) .
```

The algorithm should interpret the change in the file and generate the following query.

```
DELETE { ex:Bob foaf:name "Bob"}
INSERT { ex:Bob foaf:name "Bobby"}
WHERE { ex:Bob foaf:name "Bob" }
```

The execution of this query results in the RDF graph being changed to

```
ex:Bob foaf:name "Bobby" .
ex:Bob foaf:age 32 . #this triple is untouched
```

Listing 3.1: Resulting RDF graph

The graph in Listing 3.1 is identical to the graph that would have been created if the entire graph had been rebuilt using the new template instances.

Formal definition

Let $\mathcal{L}$ be a template library consisting of OTTR templates. Let $\mathcal{I}$ be a set consisting of instances over $\mathcal{L}$. We assume that both $\mathcal{L}$ and $\mathcal{I}$ are syntactically correct. Let the graph $\mathcal{G}$ be the result of the expansion of $\mathcal{I}$ over $\mathcal{L}$. Let $\mathcal{I}'$ be a changed version of $\mathcal{I}$ (inserting, deleting, or changing instances). Let $\mathcal{G}'$ be the graph computed as a result of the expansion of $\mathcal{I}'$ over $\mathcal{L}$.

Given $\mathcal{L}, \mathcal{I}, \mathcal{I}', \mathcal{G}$, create a set of SPARQL update queries $\mathcal{Q}$ such that the result of executing $\mathcal{Q}$ on $\mathcal{G}$ results in $\mathcal{G}'$.

Scope

The scope of this thesis will cover an algorithm that solves the presented update problem efficiently, as well as the detection of changes in the OTTR data. To achieve the objective of this thesis, we do not have to create a robust solution but rather a sufficient solution for providing results for evaluation.

We make the following simplifying assumptions:

1. The OTTR data is valid and does not contain changes that lead to inconsistently typed instances.
2. The order of the instances in the new and old instance files are the same, except for any changes that have been made.
3. The original triplestore is the expansion of the old OTTR data and is not changed by a reasoner or modified in any way.
4. No blank nodes are passed as an argument to an instance in the instance files. This assumption is discussed in Section 5.1.

Evaluation

There are various ways to assess the result of the solutions presented in this thesis that show different aspects of the solutions. As one of the goals of this thesis is to evaluate whether it would be advantageous to extend OTTR with updates, we chose to evaluate the results on how it would perform practically rather than theoretically. We will evaluate the solutions by benchmarking.

For the update algorithm to be used in a practical setting, it must outperform the current implementation in most typical use cases. Given an existing RDF graph and a change, we will measure the time it takes to detect the changes, create the update, and execute the update query. The results will be compared to a baseline, which is the time OTTR currently uses to handle updates. The benchmark will measure the time spent in multiple testing scenarios with varying graph sizes and changes as well as a varying number of duplicates or blank nodes.

Part II

Solutions

Chapter 4

Current solution: Rebuild solution

To evaluate a given solution, it is necessary to compare it to the current way of updating an OTTR-constructed triplestore. As it stands today, updating is possible but inefficient.

The current method of updating an OTTR-constructed triplestore consists of rebuilding the whole graph and replacing the existing data in the triplestore with the newly constructed graph. More precisely: given a set of instances $\mathcal{I}$ over a template library $\mathcal{L}$, a graph in a triplestore $\mathcal{G}$ resulting from expanding $\mathcal{I}$ over $\mathcal{L}$, then if a change has been made to $\mathcal{I}$, now $\mathcal{I}'$, we find the new graph $\mathcal{G}'$ resulting from expanding $\mathcal{I}'$ over $\mathcal{L}$, and replaces $\mathcal{G}$ with $\mathcal{G}'$ in the triplestore. We will refer to this existing update implementation as the *Rebuild solution*.

For most practical use cases, this method of updating is inherently slow. Regardless of the number of affected instances, all instances must be expanded and inserted into the triplestore. However, this solution is guaranteed to be correct regardless of the type of change.

As the *Rebuild solution* is very simple in nature, we can expect it to scale predictably with varied input. Rebuilding expands all instances and overwrites the existing triplestore. From this, it follows that the solution grows linearly as the number of instances increases. Both changed, and unchanged instances are treated precisely the same by expanding and overwriting the existing triplestore. This means that as long as the number of instances is constant, can we expect the *Rebuild solution* to use the same amount of time regardless of the number of changes. In other words, it scales constantly with the number of changes.

Chapter 5

Solution assessment

In this thesis we want to create an efficient update algorithm. For this reason, evaluating the solutions created through this thesis is important. In this chapter, we will first argue what a typical use case of OTTR looks like. Based on this, we will describe the characteristics of the test data and how testing is performed.

5.1 The typical use case

To evaluate the performance of our solution under realistic conditions, we need to establish some general assumptions about the typical usage patterns of OTTR. These assumptions include the total number of instances, the number of changes, the types of changes, and other relevant factors that may affect the design and performance of our solution. This section will present and justify these assumptions based on existing information and statements from industry users.

OTTR is designed to improve the efficiency and quality of the building, using, and maintaining datasets [46]. It is possible to use OTTR for small datasets, but the benefits of abstraction will be far greater with medium to large datasets. OTTR templates allow quick and easy creation of bigger RDF patterns through template instances, which benefits larger graphs. We will focus on a use case with a large dataset since OTTR is already quick enough to update small datasets in a reasonable time. OTTR's current update solution is simply to rebuild the whole RDF graph, as shown in Chapter 4. This approach works well as long as the dataset is small, but will be very time-consuming with large datasets. Therefore, our update algorithm will be intended for applications where the RDF graph is of considerable size. It is neither useful nor necessary for updating very small graphs.

A typical case where the dataset becomes very large is when it is used as a database, that is, for the purpose of storing large amounts of data. An ontology can be separated into two parts, ABox and TBox [2]. The ABox contains facts about individuals and things, while the TBox contains statements that define the relationship between classes and properties. When stating that a dataset can be used as a database, we imply that it is a dataset with an empty TBox. On the other hand, we have ontologies used for reasoning, like Owl ontologies [39], that typically have a large TBox.

Blank nodes are resources without identifiers and are generally used to imply the existence of a thing without using an IRI to identify it [16]. Since we want to create large ontologies, getting information from other sources in a different format would be natural. This information could be stored in relational databases, CSV files, or other conventional ways and translated to OTTR instances with bOTTR mappings [48]. Relational databases, Excel, and CSV files typically do not have any concept of blank nodes; thus, it would be unnatural to add unidentifiable information. Therefore, we can assume that no blank nodes will be given as an argument to an instance in an OTTR instance file. However, it is important to note that it is still possible for an instance to have a blank node as an argument, but only as a result of expanding another instance, i.e., the blank node is created by a template.

In databases, efficient use of space is important. Therefore, it is common to limit the number of duplicates and superfluous data by normalizing the database [14]. As such, we can assume that the OTTR instances created from data in other databases, will create a relatively low number of duplicate triples. In other cases where the ontology is used for reasoning, an OTTR representation might create many duplicates. OTTR templates for reasoning ontologies will usually contain TBox statements that are less dependent on the arguments. Thus, different OTTR instances of the same template might create duplicate triples.

We also assume that the update is small relative to the entire triplestore. If the triplestore is used as a database, a large update could affect millions of triples, and it seems unlikely that such a large update would happen at once.

Aibel AS is a leading Norwegian service company in the oil, gas, and offshore wind industries [3]. From personal correspondence with Christian Mahesh Hansen, an ontology specialist at the company, we gathered the following information on Aibel's use of OTTR and graph construction in general:

- The constructed graph is large, consisting of millions of statements
- Most changes are small relative to the overall size of the graph.
- Instances creating duplicated statements are rare

- Blank nodes occurring in templates are mostly used for making TBox-statements
- No blank node is passed as an argument to a user-specified instance

5.2 Testing

A series of tests is performed in which the elapsed time of a given solution is measured and compared with the elapsed time of a baseline solution; the *Rebuild solution*. The solutions are timed using the built-in nanoTime function from Java.

A solution may perform different actions depending on the input data, which, in turn, can affect performance. Therefore, having various sets of test data with different characteristics is important. We have identified three cases that potentially need to be handled explicitly by our algorithm; instances creating triples containing blank nodes, instances creating duplicated triples, and all other instances. We will use the terms *blank triples*, *duplicated triples*, and *normal triples* for triples containing blank nodes, triples that are duplicates, and all other triples, respectively. An instance that creates one or more *blank triples* will be referred to as a *blank instance*, while one creating one or more *duplicated triples* is referred to as a *duplicate instance*. If an instance creates only *normal triples*, we call it a *normal instance*. It is possible for an instance to create both a blank triple and a duplicate triple, in which case, the instance is both a blank and a duplicate instance.

For each solution, we limit the tests to only relevant properties, as not all properties may be relevant or applicable. In our test data, we chose to vary the following properties:

- Number of instances
- Number of changes, which can include:
  - *Normal instance* deletions
  - *Normal instance* insertions
  - *Duplicate instance* deletions
  - *Duplicate instance* insertions
  - *Blank instance* deletions
  - *Blank instance* insertions

As we are creating an algorithm to handle changes efficiently, investigating how the size of the change affects performance is natural. Further, as we

discussed in Section 5.1, we assume the total number of instances is large. For this purpose, seeing how the algorithm performs with an increasing number of instances is also important. Lastly, we have identified *blank instances*, *duplicated instances*, and *normal instances* as points of interest. Therefore, we will investigate how a varying number of each instance type affects the performance.

By looking at how different properties affect the result of different solutions, we get a general sense of whether a solution is viable. Also, it helps us to assess whether one technique is better than another. And lastly, it helps with understanding what assumptions must be in place for a solution to outperform the *Rebuild solution*.

Our datasets are based on the exoplanets dataset created from the *Extrasolar Planets Encyclopaedia* [52]. The exoplanets dataset contains 4678 instances of the same `Planet` template. The `Planet` template creates up to 5 triples, depending on the arguments provided. Listing 5.1 shows an excerpt of the test data, including the template and one instance.

```
1  # Planet template
2  ep:Planet [
3      ottr:IRI ?iri,
4      xsd:string ?name,
5      ottr:IRI ?star,
6      ? xsd:string ?starName,
7      ? xsd:decimal ?mass
8  ] :: {
9      o-rdf:Type(?iri, ex-o:Planet),
10     o-rdfs:Label(?iri, ?name),
11     ottr:Triple(?iri, ex-o:orbitsStar, ?star),
12     o-rdfs:Label(?star, ?starName),
13     ottr:Triple(?iri, ex-o:hasMass, ?mass)
14 } .
15
16 # Instance
17 ep:Planet(ex:XTE_J1751-305_b, "XTE J1751-305 b",
      ex:XTE_J1751-305, none, "27.0"^^xsd:decimal) .
```

Listing 5.1: excerpt of the test data

We have created a Python program to create the test data. For every test case, it creates a new and old instance file based on the "seed," which is the exoplanets dataset. This way, we can create test files with precise knowledge of the number of instances, changes, blank nodes, and other properties. If we wish to test with more than 4678 instances, more instances are created by creating synthetic `Planet` instances with random arguments.

The test program, including the implementation of all solutions tested in

this thesis, is available on GitHub [33]. The repository is licensed under the GNU Lesser General Public License v2.1.

Hardware

All results presented in this thesis were tested using the Asus laptop described below.

- Laptop: Asus Zenbook UX430UN PURE4
- Processor: Intel® Core™ i7-8550U CPU @ 1.80GHz × 8
- Memory: 16,0 Gib

Program versions

The software used in this thesis is described below.

- Operating system: Ubuntu 22.04.1 LTS 64-bit
- Java: openjdk 11.0.17
- Lutra version: 0.6.13
- Apache Jena version: 4.6.0
- Apache Jena Fuseki version: 4.5.0
- Python version: 3.10.6

Chapter 6

Simple solution

We will now look at a *Simple solution* to handling updates to OTTR instances. It makes certain limiting assumptions about duplicates and blank nodes but, is much more efficient than the *Rebuild solution* in most scenarios.

The idea behind this solution is to only update the triples corresponding to new, deleted, or modified instances. This differs from the *Rebuild solution*, which updates all triples regardless of change. To do this, we look at the difference between the old and the new instance files. Then, we delete the result of expanding everything changed from the old file and insert the result of expanding everything new in the new file.

In this chapter, we will first look at a description of the proposed solution. Following this, we illustrate the solution with an example. Furthermore, we will look at what assumptions need to be made for the algorithm to work. Lastly, we discuss the results of testing. The implementation of this solution will be shown in Chapter 7.

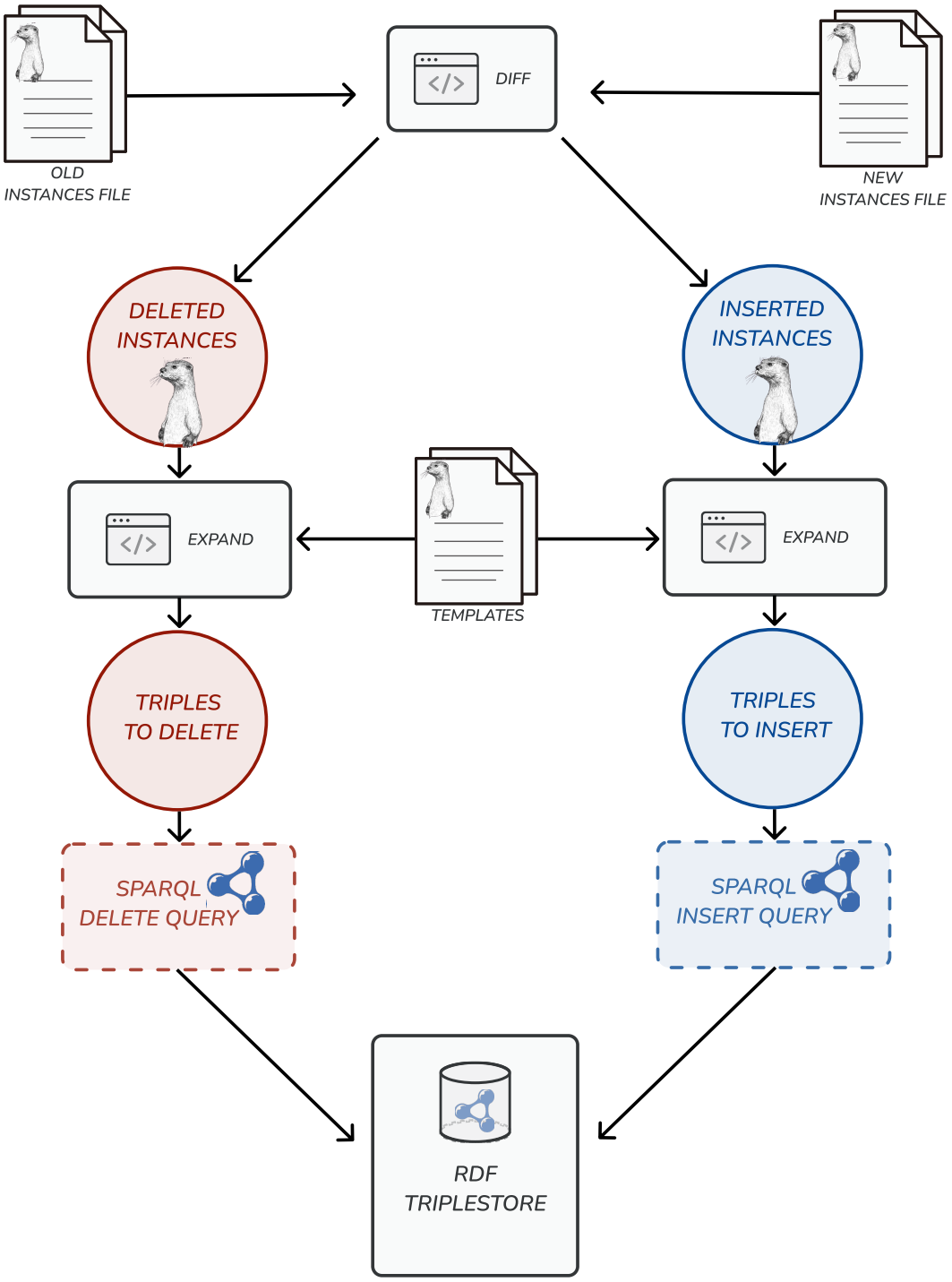

Figure 6.1: Simple solution overview

6.1 Description

The solution can be described as follows:

1. First, the diff algorithm identifies deleted instances $\mathcal{D}$ and inserted instances $\mathcal{I}$. Modified instances are interpreted as deleting the old version and inserting the new one. Meaning the old version of a triple is in $\mathcal{D}$, and the new one is in $\mathcal{I}$.
2. Then, a SPARQL delete query is created that explicitly lists all triples from the result of expanding $\mathcal{D}$.
3. Then, a SPARQL insert query is created that explicitly lists all triples from the result of expanding $\mathcal{I}$.
4. Finally, execute the two queries on the triplestore.

There is an illustrated overview of the *Simple solution* in Figure 6.1. This solution is straightforward and reads modified instances as deleting the old version, and inserting the new. This means all triples expanded from a modified instance will be deleted and possibly inserted again.

Example

Given a template

```
ex:Person[?person, ?name, ?age] :: {
    ottr:Triple(?person, foaf:name, ?name),
    ottr:Triple(?person, foaf:age, ?age)
}
```

And template instances

```
ex:Person(:bob, "Bob", 32) .
ex:Person(:ole, "Ole", 12) .
```

And a corresponding triplestore

```
:bob foaf:name "Bob" .
:bob foaf:age 32 .
:ole foaf:name "Ole" .
:ole foaf:age 12 .
```

Then we make two changes to the instances. We remove Ole and change Bob's age to 33.

```
ex:Person(:bob, "Bob", 33) .
```

Now the *Simple solution* will add the expansion of `ex:Person(:ole, "Ole", 12)` to the delete query, and it will also add the expansion of `ex:Person(:bob, "Bob", 32)` to the delete query since it has been modified. Lastly, the expansion of `ex:Person(:bob, "Bob", 33)` is being inserted, as it is the new version of Bob. We now have two queries:

```
DELETE WHERE {
:bob foaf:name "Bob" .
:bob foaf:age 32 .
:ole foaf:name "Ole" .
:ole foaf:age 12 .
}

INSERT DATA {
:bob foaf:name "Bob" .
:bob foaf:age 33 .
}
```

6.2 Assumptions

In order to make the *Simple solution* work, we will make certain simplifying assumptions. Blank nodes do not have an ID and can, therefore, not be explicitly listed in a SPARQL query. This creates problems for the *Simple solution*. We can insert blank triples as usual, but deleting becomes challenging as we cannot identify the triples we want to delete. We split the blank nodes into two groups; *top-level blank nodes* and *local blank nodes*. Top-level blank nodes are blank nodes that are passed as arguments to an OTTR instance in the instance file (top-level instance), which means it is not created during expansion of a template. A local blank node, however, is created during expansion of a template. In triplestores used as databases, it makes little sense to put data that lack identifiers as we do not know what they refer to, therefor we make a simplifying assumption that top-level blank nodes do not exist.

We believe local blank nodes occur more frequently, for example when grouping resources such as all parts of an address. Thus, it is in our interest to allow local blank nodes, which we will look at in Chapter 9. For now, we assume that blank triples cannot be deleted, to make the *Simple solution*

work. We call this assumption the *local blank node assumption*, as we already have assumed the absence of top-level blank nodes. This means blank instances cannot be deleted or modified, as it would result in deleting a blank triple. This assumption is narrow, and it might be hard to recognize for users if this will be a problem. As a more general alternative, it is possible to assume the absence of blank nodes in general.

```
1  TEMPLATE
2  ex:Car[?brand, ?color] :: {
3      ottr:Triple(?brand, rdf:type, :Car),
4      ottr:Triple(?brand, :hasColor, ?color),
5      ottr:Triple(:Car, rdf:type, :Class)
6  } .
7
8  INSTANCES
9  ex:Car(:audi, :red) .
10 ex:Car(:skoda, :blue) .
11
12 RESULT OF EXPANDING INSTANCES
13 :audi rdf:type :Car .
14 :audi :hasColor :red .
15 :Car rdf:type :Class .
16 :skoda rdf:type :Car .
17 :skoda :hasColor :blue .
```

Listing 6.1: Synchronized triplestore

An RDF graph is defined as a set of triples, which means that there are no duplicates, and thus deleting an instance might affect triples created by other instances. An example is shown in Listing 6.1 where we see that `ottr:Triple(:Car, rdf:type, :Class)` is created from both the Audi and the Skoda. If we then delete the result of expanding the Audi instance (Listing 6.2), we would delete the only `ottr:Triple(:Car, rdf:type, :Class)` in the triplestore. But that triple should still exist, as it also is created from the Skoda instance.

Solving this problem would require retrieving knowledge of which triples are created by which instances at query time, which would require an additional data structure or another efficient way to recognize duplicates. For now, we can avoid this by assuming that a triple inserted by multiple instances cannot be deleted, and we call this the *Duplicate assumption*. We show a solution removing this assumption in Chapter 8. Although this assumption may address the problem, it is narrow, and it may be challenging for users to determine if the algorithm is suitable. As an alternative, we will propose a few more general assumptions to tackle the same issue. Although these assumptions are more restrictive, they may be easier to use.

```
# TEMPLATE
ex:Car[?brand, ?color] :: {
    ottr:Triple(?brand, rdf:type, :Car),
    ottr:Triple(?brand, :hasColor, ?color),
    ottr:Triple(:Car, rdf:type, :Class)
} .

# INSTANCES
ex:Car(:skoda, :blue) .

# RESULT OF EXPANDING INSTANCES
:skoda rdf:type :Car .
:skoda :hasColor :blue .
```

Listing 6.2: Desynchronized triplestore after deleting expansion of Audi

An alternative assumption is to only allow insertions. The duplicate problem occurs only when deleting, meaning removing all delete operations, including modifying, will also solve the problem. This assumption is more strict than the previous assumption and significantly limits possible updates, but it is straightforward to evaluate the usability.

Another alternative is something in the middle. We can assume that several instances do not create the same triple. By making this assumption, we can delete the triples corresponding to a deleted instance without risking desynchronizing the triplestore. All these alternatives will make the *Simple solution* work.

6.3 Experimental evaluation

Now we will look at the performance of the *Simple solution* and compare it to the *Rebuild solution*.

From Figure 6.2 it is clear that the *Simple solution* scales much better than the existing *Rebuild solution* when the number of changes is relatively low, 10 in this case. Figure 6.3 captures the point of intersection between the two solutions in a case with 4 changes. Already at 7 instances, the *Simple solution* is faster than the *Rebuild solution*. Figure 6.4 shows that the required time seems to increase linearly as the triplestore increases in size. Looking at Figure 6.5 where we vary the number of changes rather than instances, we see that the *Simple solution* scales linearly with the number of changes, while the *Rebuild solution* is constant. The *Simple solution* is getting slower in comparison to the *Rebuild solution* as the change affects bigger parts of the triplestore.

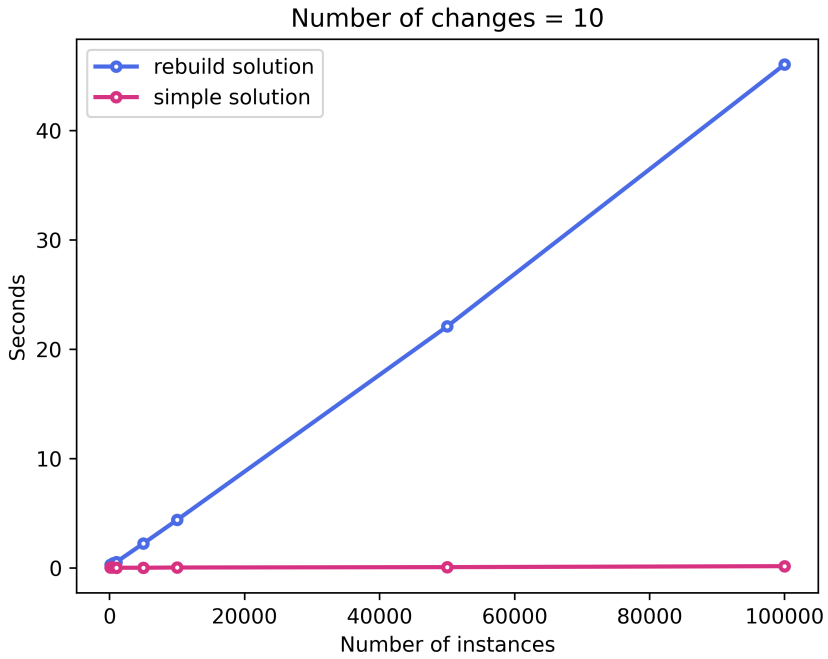

Figure 6.2: Simple solution VS. Rebuild solution

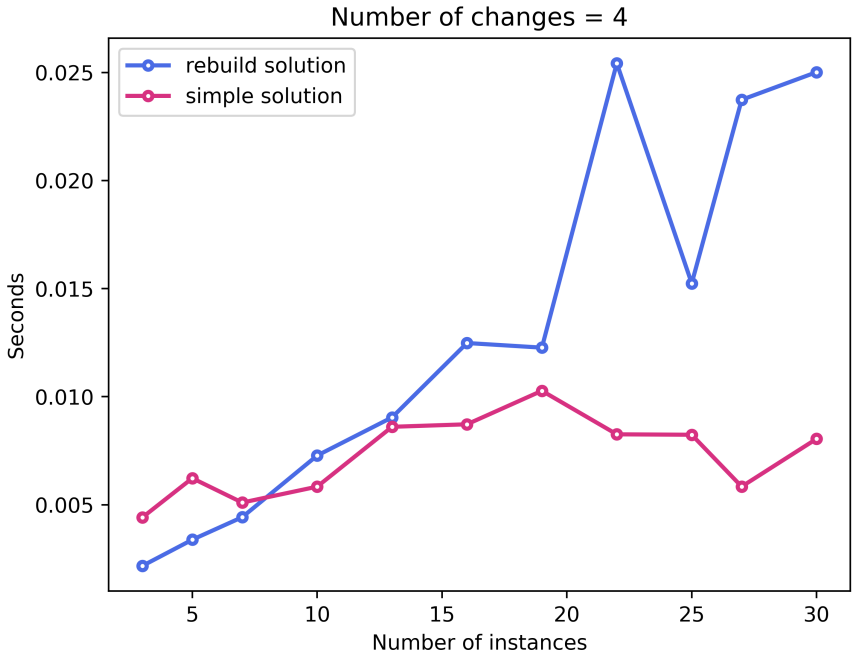

Figure 6.3: Simple solution VS. Rebuild solution point of intersection

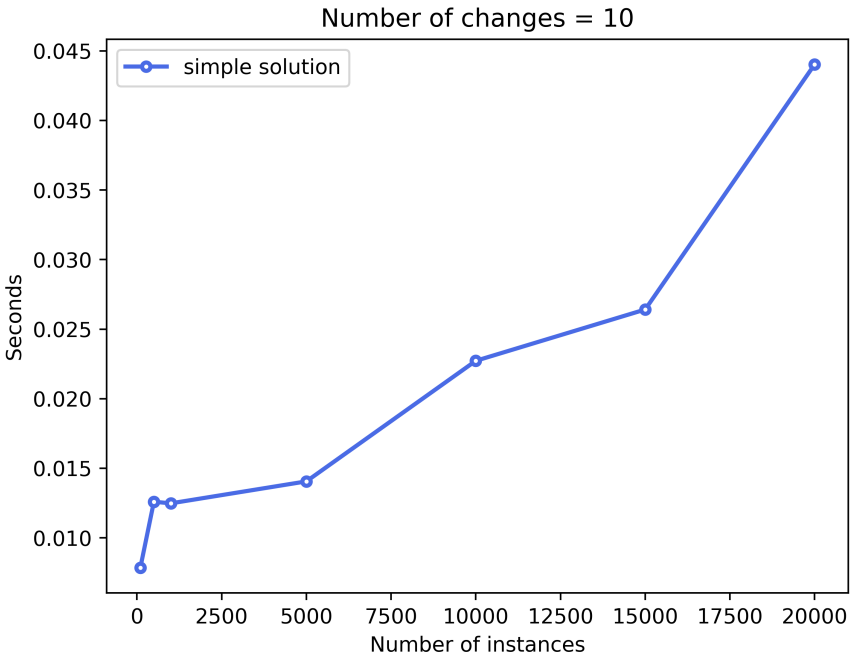

Figure 6.4: 10 changes, varying number of instances

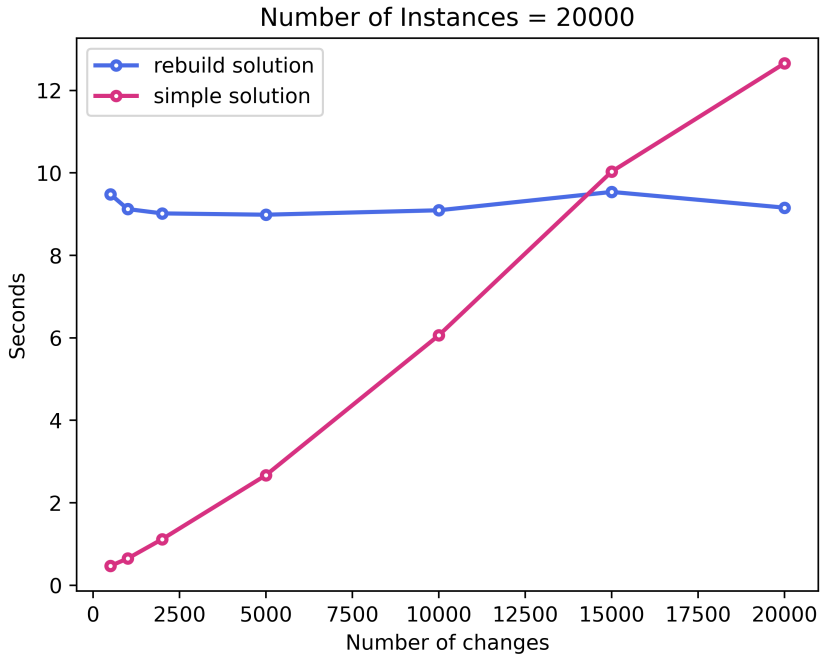

Figure 6.5: 20000 instances, varying number of changes

Figure 6.6 shows what parts of the algorithm use the most time during execution when the number of instances is relatively high (100000), and the number of insertions and deletions are high (500). Figure 6.7 illustrates a similar example where the number of insertions and deletions are few; 10 deletions and 10 insertions. We can see that expanding instances takes longer as the number of changes increases, while the diff part is consistently long when the number of instances is this high. The query part also takes longer with larger changes.

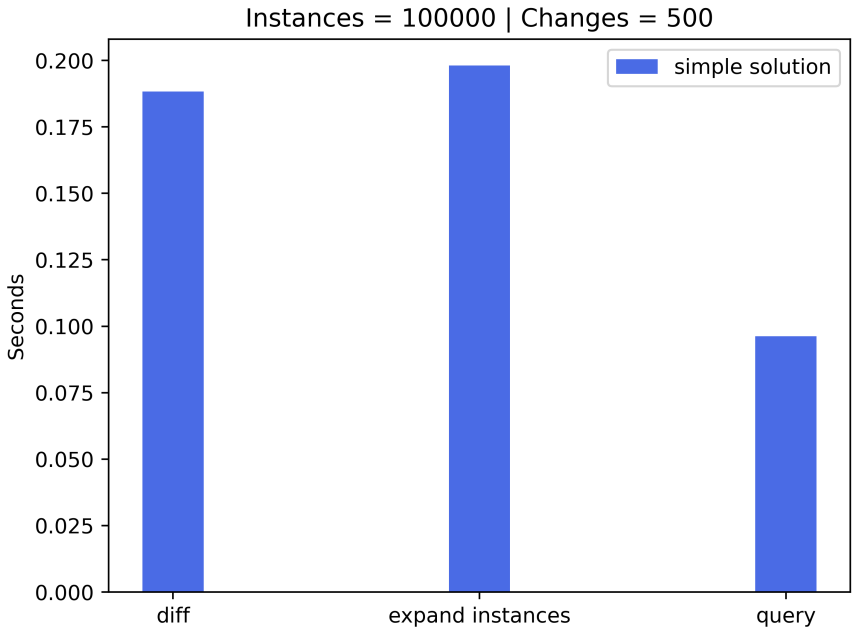

Figure 6.6: 100 000 instances, 250 deletions, 250 insertions

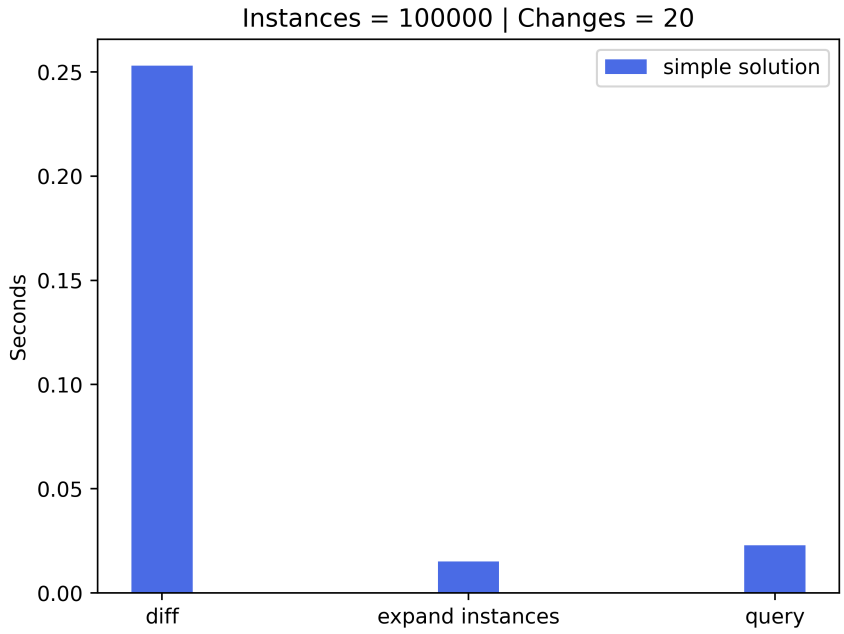

Figure 6.7: 100 000 instances, 10 deletions, 10 insertions

6.4 Discussion

The *Simple solution* shows an excellent improvement over the *Rebuild solution* when the number of changes is relatively low. Compared to the *Rebuild solution*, it performs much better with larger triplestores, which is how OTTR is typically used.

From Figure 6.5 we see that the *Simple solution* scales linearly with respect to the number of changes, while the *Rebuild solution* scales constantly. Although the *Rebuild solution* scales better with the number of changes than the *Simple solution*, we see that in Figure 6.5 that the constant time spent by the *Rebuild solution* is large. It was not until 15000 out of 20000 instances changed that *Rebuild solution* was faster. Nevertheless, this is an unrealistic scenario as it would require changing a considerable number of instances simultaneously.

In Figure 6.4, we see that the *Simple solution* scales linearly with respect to the number of instances. In Figure 6.2, it is clear that also the *Rebuild solution* scales linearly with the number of instances. Still, the *Simple solution* heavily outperforms rebuilding. This makes sense as the *Simple solution* scales linearly because of the diff operation, and the *Rebuild solution* because of the expanding operation. Looking through a file for differences is cheaper than expanding all instances. From this, we can conclude that the larger the difference is between the number of changes and the total

number of instances, the better the *Simple solution* performs compared to the *Rebuild solution*. If the difference becomes too small, rebuilding will be most efficient.

The *Simple solution* handles a change to an instance by deleting the entire expansion of the old instance, followed by inserting the entire expansion of the new instance. This can cause a few triples to be deleted and inserted unnecessarily. This should not be noticeable as long as the templates are not deeply nested or each template expands to relatively few other instances. However, in an extreme case where one instance expands to very many triples and a tiny change is made so that only one triple change would be sufficient, we can expect the a worse performance from the *Simple solution*.

From Figure 6.6 and Figure 6.7, we see that the number of changes mainly affects the expansion of instances. Also, the difference algorithm is the most time-consuming part with small changes. We also know from Section 5.1 that a typical use case involves relatively few changes to a big triplestore. This means the expansion of changed instances will not be very time-consuming in the typical case. Thus, it does not matter exactly how many triples the changed instances expand to. Therefore, we will not perform tests with different template sizes, as we do not expect them to greatly impact the results.

We have made strict assumptions for the *Simple solution* to work; blank nodes and duplicates cannot be deleted. These assumptions will restrict the practical use of the algorithm a lot. It might also be difficult for a user to ensure that no duplicates or blank nodes are deleted. This will be easier if we use a more general assumption, such as allowing only insertions. It should be easy for a user to judge whether the insertion operation is sufficient, and if that is the case, the *Simple solution* is a good choice.

The *Simple solution* performs very well but has limited applicability due to its strict assumptions. In the following chapters, we will find other solutions without these assumptions rather than focusing on improving the efficiency of the *Simple solution*. In Chapter 9, we will show how the local blank node assumption can be removed, and in Chapter 8, we will show how the duplicate assumption can be removed.

Chapter 7

Implementation

We have implemented a program that runs and tests different solutions. This section provides an overview of the program's general structure and explains our choices. Additionally, we will look at the implementation of the *Simple solution*. The implementation of other solutions will be explained in their respective chapters.

An update algorithm in OTTR would be implemented as a part of Lutra, meaning it would run on the Java virtual machine. We chose to implement our solutions as a Java program to ensure our testing has the same characteristics as an actual implementation. We concluded that it would be unnecessary to implement it directly into Lutra, as the extra time spent integrating would not give better results. As such, the program is implemented as a standalone application that reads the input data from the disk and queries the triplestore directly. Lutra is imported as a library to read templates and expand instances. We run the triplestore as its own process on the same computer as the update program. Communication between the program and the triplestore is done over HTTP on the localhost.

The testing program has the following characteristics:

1. Runs similarly to a possible implementation in Lutra
2. Easy to implement and test new solutions
3. Is able to swap solution algorithms

The general overview of the program is illustrated in Figure 7.1. The program reads the template files and the old and new instance files. The implementation is created as a collection of services and a set of solutions that use these services. The solutions query a local Fuseki triplestore to perform the updates.

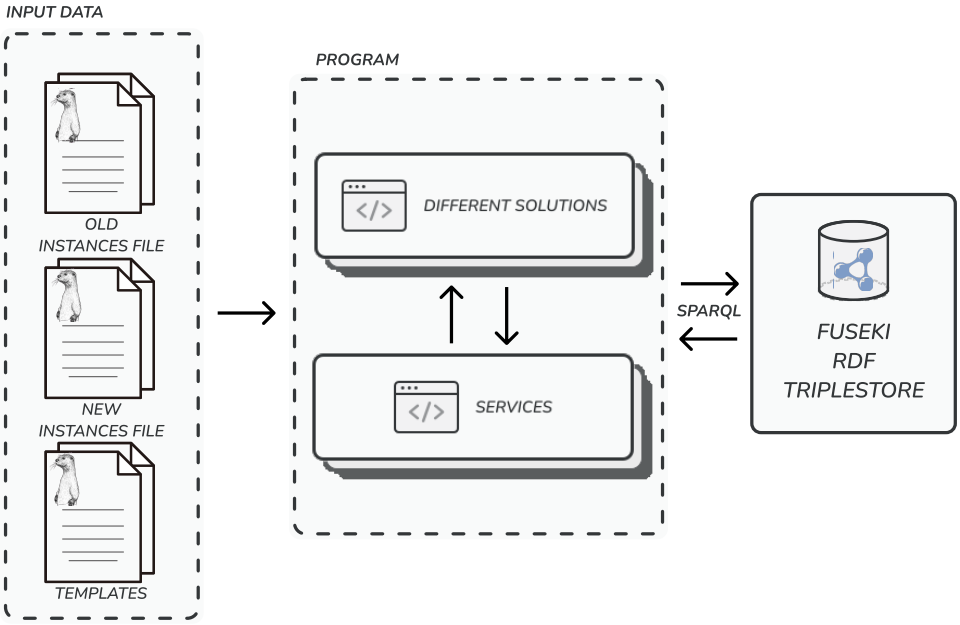

Figure 7.1: Program overview

A more detailed look at the program outline is shown in Figure 7.2.

We have used the Apache Jena framework [13] to work with RDF. Apache Jena is Java-based and supports different RDF formats, as well as different stream operations that are useful when working with large amounts of data. Apache Jena also offers a SPARQL server, Apache Jena Fuseki, that we can host locally and communicate with over HTTP. Apache Jena is popular and open source, making it a natural choice. Other possible options are RDF4J [51] and dotNetRDF [24].

7.1 Simple solution implementation

We have implemented a single module for the *Simple solution*. The path to the files with old and new instances is passed as arguments. We use a service with the diff algorithm to get the sets of deleted and inserted instances. Then, these sets are passed to an OTTR service for expansion, which returns new and old triples that are put into Jena models [35]. We pass the models as arguments to Jena updatebuilders [54] to create the two SPARQL queries. The queries are sent to the triplestore over HTTP.

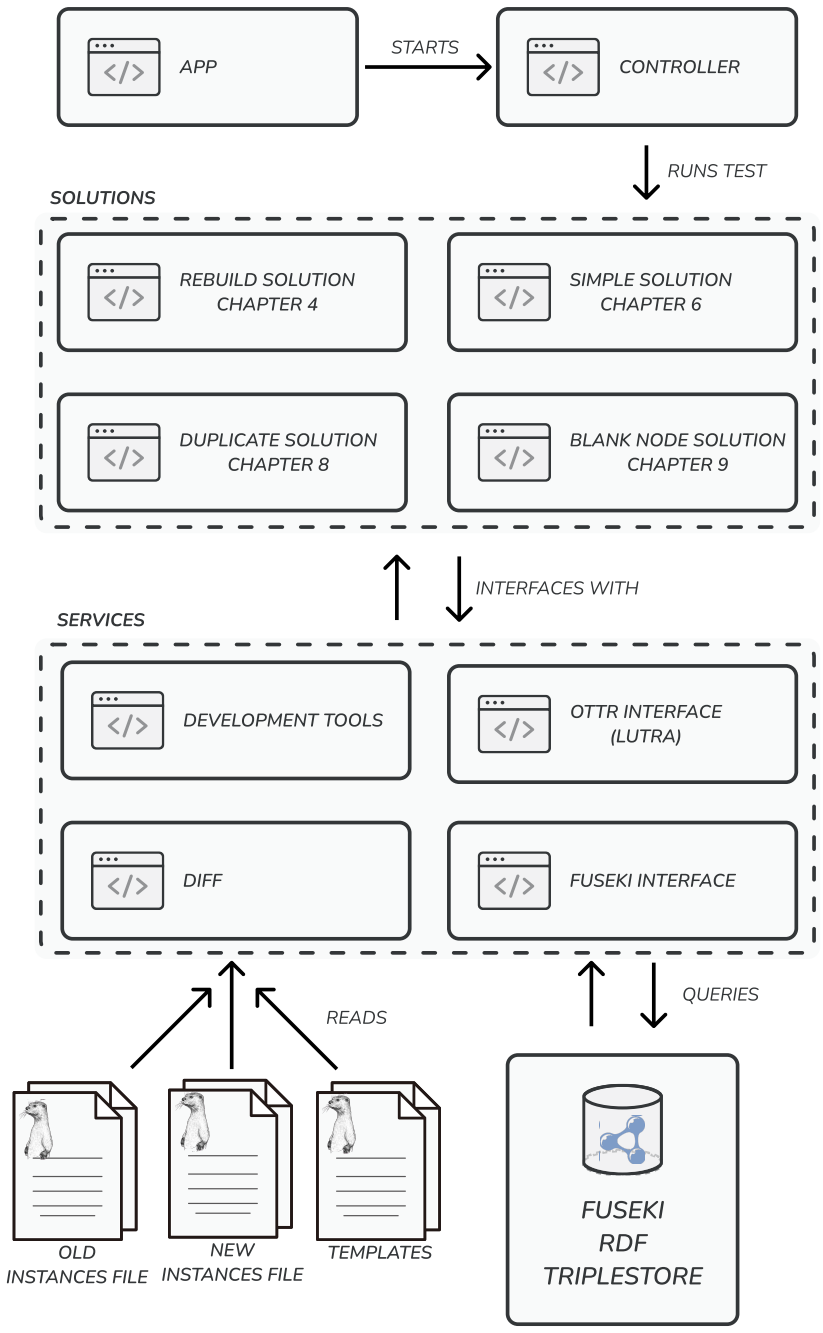

Figure 7.2: Detailed overview

Chapter 8

Removing the duplicate assumption

The duplicate assumption used in the *Simple solution* is very restrictive for some kinds of datasets, especially ontologies used for reasoning. It might also be difficult for a user to guarantee that no duplicates will be deleted. This chapter will discuss different approaches to removing the duplicate assumption and what new considerations this would entail. We will start by showing how the problem occurs. Then, we will look at a way to enable duplicates, followed by the implementation of that approach. After that, we will discuss some alternative approaches, and lastly, we will present and discuss the results.

The duplicate assumption is defined as follows: "A triple inserted by multiple instances cannot be deleted". A triplestore is a set of unique triples, meaning no duplicates exist [56]. However, multiple OTTR instances can expand to the same RDF triple. If several instances create identical triples, only one will appear in the triplestore. Listing 8.1 outlines a typical case where duplicates occur.

```
1 ex:subClassOf[?child, ?parent] :: {
2     ottr:Triple(?child, :subClass, ?parent),
3     ottr:Triple(?child, rdf:type, :Class)
4 }
5
6 ex:subClassOf(:A, :B) .
7 ex:subClassOf(:A, :C) .
```

Listing 8.1: Template and Instances creating a duplicate

We see that the instance on line 6 and the instance on line 7 in Listing 8.1 both create the same triple `(:A rdf:type :Class)`. Expanding the instances and inserting them into a triplestore will result in only one of

the two duplicated triples existing. The problem occurs when we want to delete an instance. If we now delete the instance at line 7, then we have to delete the triple `(:A rdf:type :Class)` if we follow the *Simple solution*. Doing this leaves us with the instance at line 6 not being synchronized with the triplestore as a triple is missing. The duplicate assumption avoids this problem, but restricts the use of the algorithm.

8.1 Counting duplicates in a separate graph

One way to enable deletion of duplicate instances is to maintain a record of the number of duplicates for each triple. By doing this, we can decrement the number of duplicates instead of deleting the original triple. We will use a separate graph in the triplestore to store the duplicates.

Assume that we have a graph $\mathcal{G}$ in a triplestore containing the expansion of the OTTR instances. We can then create a new graph $\mathcal{C}$ to contain triples keeping track of the number of duplicates. It is also possible to use only one graph, but we believe it is better to separate data and metadata, to avoid metadata affecting queries over the default graph. We let the triples in $\mathcal{C}$ have the form described in Listing 8.2 and refer to them as *counter triples*. A triple t can have a corresponding counter triple, which we will call ct. We use RDF-star as an unambiguous way to refer to a triple [31].

```
1 <<original triple>> ex:count 2 .
```

Listing 8.2: Counter triple

In the following algorithm, we let:
t=triple, ct=counter triple, $\mathcal{G}$=default graph, $\mathcal{C}$=counter graph

Inserting a new triple t into the triplestore:

1. IF $t \notin \mathcal{G}$, THEN
   (a) Insert t into $\mathcal{G}$
2. ELSE IF $ct \in \mathcal{C}$, THEN
   (a) Increment ct count by 1
3. ELSE insert ct into $\mathcal{C}$ with count 2

Deleting a triple t:

1. IF $ct \in \mathcal{C}$, THEN
   (a) Decrement ct count by 1
   (b) IF ct count is 1, THEN
      i. Delete ct from $\mathcal{C}$
2. ELSE
   (a) delete t from $\mathcal{G}$

When inserting a triple t, as described in the algorithm above, we start by checking if the triple exists in the default graph $\mathcal{G}$. If it does not, we can insert t into $\mathcal{G}$, and we are done. If it does already exist, we have a duplicate. We cannot add duplicates in $\mathcal{G}$, so we use a counter triple representation, ct of t, in the counter graph $\mathcal{C}$. ct keeps track of how many duplicates there are of t. If ct does not already exist in $\mathcal{C}$, we insert it with the count 2. If it already exists, we increment the count of ct by one. In Listing 8.3, we see the result of using this method for insertion combined with deletion.

Deleting a triple t is fairly similar to inserting a triple. First, we check if a counter triple, ct, exists in $\mathcal{C}$. If ct does not exist, we know t is not a duplicate, and the triple can be deleted from $\mathcal{G}$. If ct exists, we decrement its value. If the decremented count is 1, we delete ct as there are no duplicates. In Listing 8.3, we see the result of deleting `(<a> <b> <c>)` and inserting `(<d> <e> <f>)` when there exists a counter triple for both triples in the default graph. The counter triple of `(<a> <b> <c>)` is decremented by one and is removed as the counter value is 1. `(<a> <b> <c>)` will still remain in the default graph. `(<d> <e> <f>)` is incremented from 3 to 4.

```
1  # BEFORE QUERY
2  # default graph
3  <a> <b> <c> .
4  <d> <e> <f> .
5  # counter graph
6  << <a> <b> <c> >> ex:count 2 .
7  << <d> <e> <f> >> ex:count 3 .
8
9  # AFTER DELETE <a> <b> <c> and INSERT <d> <e> <f>
10 # default graph
11 <a> <b> <c> .
12 <d> <e> <f> .
13 # counter graph
14 << <d> <e> <f> >> ex:count 4 .
```

Listing 8.3: Counting duplicates example

Duplicates within the update

So far, the implementation does not handle duplicates within the update, such as when two equal triples should be deleted simultaneously. In that case, the goal would be to decrement the counter triple by 2 instead of 1. The problem is that the expansion of the OTTR instances results in a set without duplicates. This is solved by expanding the instances one at a time and counting the occurrences in a hashmap, we use 2 occurrences as an example. We now know how much we should increment or decrement the counter triple. We will now modify the algorithm to handle larger increment and decrement operations.

Deleting n equal triples:
given:
t=triple, ct=counter triple, $\mathcal{G}$=default graph, $\mathcal{C}$=counter graph

1. IF $ct \in \mathcal{C}$, THEN
    (a) Decrement ct count by n
    (b) IF ct count < 2, THEN
        i. Delete ct from $\mathcal{C}$
    (c) IF ct count < 1, THEN
        i. Delete t from $\mathcal{G}$
2. ELSE
    (a) delete t from $\mathcal{G}$

If we are inserting multiple equal triples, we modify the insertion algorithm in the same way. We make a hashmap after expansion to find the number of insertions for each triple. We insert the triple in the default graph if it does not exist, then update the counter graph with the number of duplicates.

Inserting n equal triples:
t=triple, ct=counter triple, $\mathcal{G}$=default graph, $\mathcal{C}$=counter graph

1. IF $t \notin \mathcal{G}$, THEN
    (a) Insert t into $\mathcal{G}$
    (b) IF $n > 1$, THEN
        i. Insert ct in $\mathcal{C}$ with count n
2. ELSE IF $ct \in \mathcal{C}$, THEN
    (a) Increment ct count by n
3. ELSE
    (a) insert ct into $\mathcal{C}$ with count n+1

Duplicates within the expansion of an instance

The modified update algorithm now handles duplicates within the update by expanding one instance at a time and storing the occurrences. However, duplicates may also occur when expanding a single instance. When expanding OTTR instances, the result is a stream of triples, which is read and put in a Jena model. The same triple might occur several times in the stream, but the Jena model is a set of triples, which by default is without duplicates. This means duplicates from the same instance will be lost during expansion, even when expanding only one instance at a time. However, this is not a problem for our algorithm, as shown in the next paragraph.

Let us say we have an instance $\mathcal{I}$ that expands into a multiset of triples containing n occurrences of the same triple t where $n > 1$. All solutions described in this thesis handle instances as a whole, either deleting or inserting the whole instance. From this, it follows that every operation done on $\mathcal{I}$ affects all n duplicates simultaneously. In other words, the counter for t can only be changed in increments of size n when $\mathcal{I}$ is changed. We can never delete one of the n duplicates without deleting the rest, and likewise with insert. If $\mathcal{I}$ creates t, then it does not matter how many duplicates there are. Thus, we can treat all n duplicates from $\mathcal{I}$ as one triple, which is what our solution already does.

Implementation

We have now shown the idea behind the *Duplicate solution*, and we will now discuss the implementation and show how the SPARQL queries are computed.

The implementation of the *Duplicate solution* is similar to the *Simple solution*. The solution is in a single module that communicates with different services. OTTR-service for expansion of instances, Diff-service for identifying changes, and Fuseki-service for querying the triplestore. We use Jena models to represent sets of triples, and we use UpdateBuilders to create SPARQL queries. However, the *Duplicate solution* uses RDF* and SPARQL* when querying the counter graph. Apache Jena Fuseki supports RDF* and SPARQL* by default [11]. We will now take a closer look at the implementation of insertion and deletion with duplicates allowed. Figure 8.1 shows an illustrated overview of the implemented algorithm.

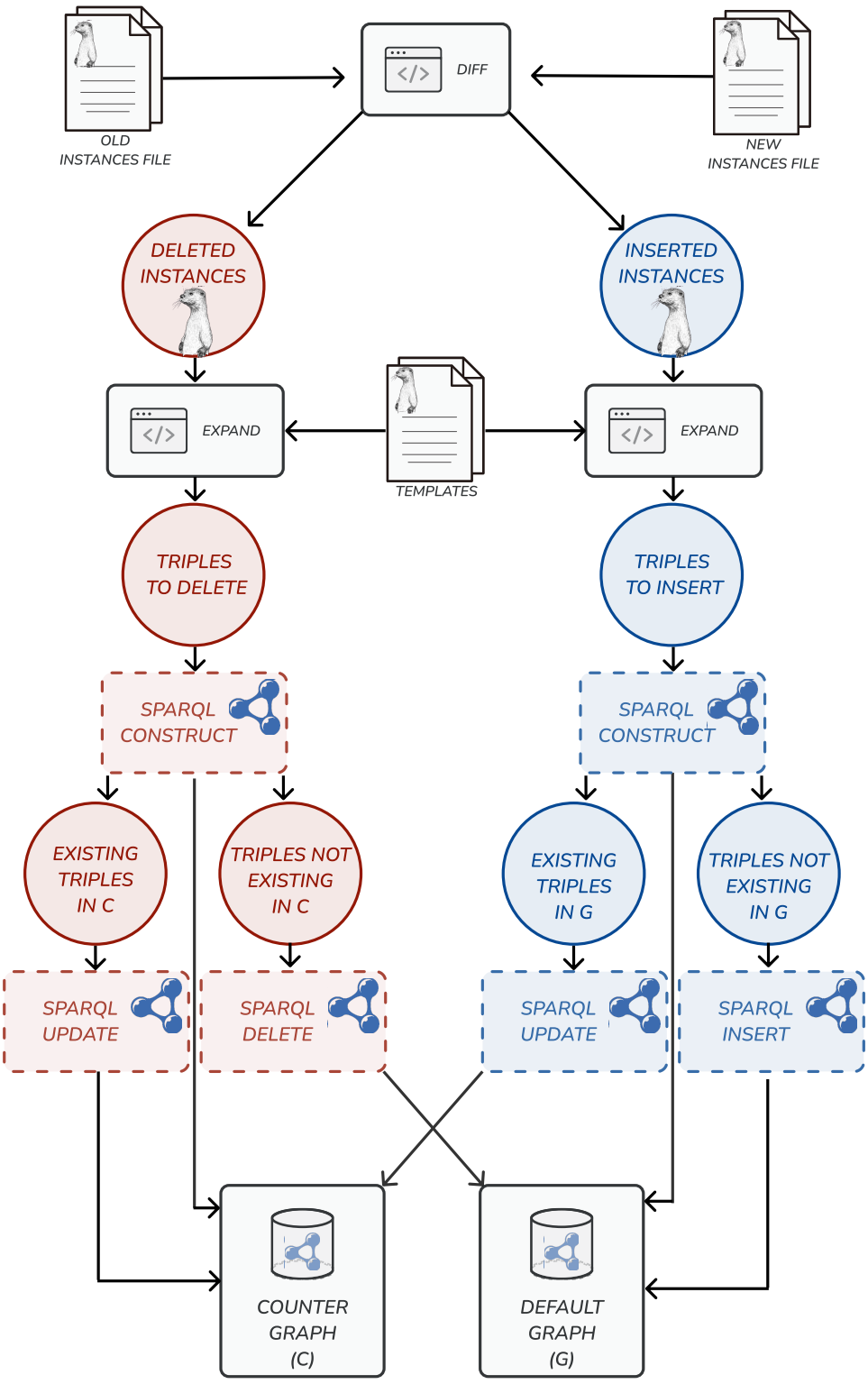

Figure 8.1: Duplicate solution overview

Insertion

Insertion starts with checking if t is already in $\mathcal{G}$. We want to check this with all the triples we are inserting simultaneously. A `CONSTRUCT` query allows us to return a graph that contains all the matching triples. We can use `VALUES` to specify the triples we are looking for. Listing 8.4 shows a query looking for three triples. If only `(:kari :livesIn :Norway)` exists in $\mathcal{G}$, then a graph with that triple is returned.

```
1  CONSTRUCT {
2      ?subject ?predicate ?object
3  }
4  WHERE {
5      ?subject ?predicate ?object .
6  }
7  VALUES (?subject ?predicate ?object) {
8      (:kari :livesIn :Norway)
9      (:car :hasOwner :kari)
10     (:car :hasColor :red)
11 }
```

Listing 8.4: Batch query looking for any triple in VALUES

We start by asking for all triples to insert, $\mathcal{I}$, in the `CONSTRUCT` query. The result of the `CONSTRUCT` query gives us a graph with the existing triples, e. By doing set difference between $\mathcal{I}$ and e, we get the non-existing triples, $\neg e$. The triples in $\neg e$ can be inserted into G with a normal `INSERT` query as they do not already exist. The existing triples e will have at least one duplicate, so we look at the counter graph $\mathcal{C}$. We can target the counter graph by using a `WITH` clause. First, we check if there is a counter triple ct for each triple in e. We use the `VALUES` clause to define the possible values for the `?subject` variable, and these are the triples from e. In Listing 8.5, this is done with `(:kari :livesIn :Norway)` as the only triple in e, but we could add several triples to the values clause. Then we use `OPTIONAL` to search for the counter triple ct for every defined subject value. If ct exists, `?old_count` will be bound. If `?old_count` is bound, `?new_count` will be bound to `?old_count` +1. Then we are incrementing the counter with `DELETE` and `INSERT`. Otherwise, if ct does not exist, `?new_count` is bound to 2, creating a new counter triple. We can increment it by more than 1 if we insert several duplicates of `(:kari :livesIn :Norway)`.

```
WITH <:counterGraph>
DELETE {
    ?subject :count ?old_count .
}
INSERT {
    ?subject :count ?new_count .
}
WHERE {
    VALUES ?subject {
    << <:kari> <:livesIn> <:Norway> >>
    }
    OPTIONAL {?subject :count ?old_count}
    BIND(if(bound(?old_count), ( ?old_count + 1 ),
       2) AS ?new_count)
}
```

Listing 8.5: Insertion, increment counter

Deletion

Deleting a collection of triples t starts with checking if corresponding counter triples ct exist in $\mathcal{C}$. This is done with a `CONSTRUCT` query similar to the insertion in Listing 8.4, but with counter triples in $\mathcal{C}$ instead. The result is a set of existing counter triples e. By doing set difference we get the set of not counted triples $\neg e$. Every t in $\neg e$, is deleted from $\mathcal{G}$ with a normal `DELETE` query. For every ct in e, the count value of ct should be decremented and deleted if below 2. This is done by deleting the old version of ct and inserting it again with an updated count as long as the updated count is greater than 2. The query is very similar to the insertion. In Listing 8.6, we see an example where we want to delete `(:kari :livesIn :Norway)`. We bind `?new_count` to `?old_count` -1 to decrease the count value. Note that we can decrement it by more if we delete several duplicates of `(:kari :livesIn :Norway)`. SPARQL does not allow us to always delete a triple while having a conditional insertion of the new version. Thus, we must remove the counter triples with a count less than 2 in a separate query in Listing 8.7. We can also decrease the count value with more than 1, if there are more duplicates within the update. Then it is possible to have a count value of 0, which means we must delete t from $\mathcal{G}$ as well.

This is the tested implementation of the *Duplicate solution*. The results will be shown and discussed later in Section 8.3. However, we will first show some alternative approaches to this solution.

```
WITH <:counterGraph>
DELETE {
    ?subject ?predicate ?old_count
}
INSERT {
    ?subject ?predicate ?new_count
}
WHERE {
    ?subject ?predicate ?old_count
    FILTER (?subject IN (<< <:kari> <:livesIn>
       <:Norway> >>))
    BIND(?old_count - 1 AS ?new_count)
}
```

Listing 8.6: Deletion, decrement counter

```
WITH <:counterGraph>
DELETE WHERE {
    ?subject ?predicate ?duplicates
    FILTER(?duplicates < 2)
}
```

Listing 8.7: Deletion, clean up all counters less than 2

8.2 Alternative approaches

There are several ways to allow deletion of duplicates. We can use different data structures, different configurations, and different methods than we have previously shown. We will now show some alternatives and discuss why we did not use them as our *Duplicate solution*.

Counting all occurrences

One alternative approach to storing duplicates would be to not only keep a counter for the triples that occur two or more times, but a counter for all triples. An example of how this would look is seen in Listing 8.8. Just like the *Counting duplicates* approach, we have a default graph $\mathcal{G}$ where all normal RDF-triples exist. In this approach, however, the graph $\mathcal{C}$ keeps track of the number of occurrences of every triple, not only duplicates. Since duplicates are rare, most counter triples in $\mathcal{C}$ will have the count set to 1.

```
# BEFORE QUERY
# default graph
<a> <b> <c> .
# counter graph
<< <a> <b> <c> >> ex:count 3 .

# AFTER DELETE <a> <b> <c> and INSERT <d> <e> <f>
# default graph
<a> <b> <c> .
<d> <e> <f> .
# counter graph
<< <a> <b> <c> >> ex:count 2 .
<< <d> <e> <f> >> ex:count 1 .
```

Listing 8.8: Counting occurrences example

The steps in inserting and deleting a triple t are as follows:

The steps in inserting t:

Given:
t = triple, ct=counter triple, $\mathcal{G}$ = default graph, $\mathcal{C}$ = counter graph

1. IF $ct \in \mathcal{C}$, THEN
   (a) Increment ct count
2. ELSE
   (a) insert t into $\mathcal{G}$
   (b) insert ct with count 1 into $\mathcal{C}$

The steps in deleting t:

bellow we let:
t = triple, ct=counter triple, $\mathcal{G}$ = default graph, $\mathcal{C}$ = counter graph

1. IF $ct \in \mathcal{C}$, THEN
   (a) Decrement ct count
   (b) IF ct count is 0, THEN
      i. Delete ct from $\mathcal{C}$ and t form $\mathcal{G}$

The major drawback of using this approach is the increased space usage. This approach stores a counter version of all triples, doubling the required space. We also know that the speedup will be minimal over the original approach, as looking for t in $\mathcal{G}$ takes the same time as looking for t in $\mathcal{C}$. This

approach additionally makes the original graph redundant, as everything is in the counter graph. We could exclude the original graph to save space, but then every query would have to be adjusted for the structure of the counter triple. We consider this a possible, but unpractical approach as one will have to change the original structure of the triplestore. Thus, we will continue with the implementation of counting duplicates and not counting occurrences.

Using an external data structure

We have already seen that we can count duplicates to remove the duplicate assumption, but we do not have to store the duplicates in the triplestore. It is also possible to use an external data structure to store duplicates. One option could be to use a hash map with triples as keys and counters as values. This hash map could then be saved in a separate file. We would still have to ask the triplestore about triples in $\mathcal{G}$, which is the slow part. One could also use a hash map with the counting occurrences approach for constant time lookup, but this would require loading the entire triplestore into memory which might not be an option. Loading a large amount of data into memory can take a long time. More importantly, if the hashmap is larger than the available memory, reading data between storage and memory will be time-consuming. Other data structures can be interesting as well. However, unless we represent the entire triplestore in another way, we would still have to check with the triplestore to identify new duplicates with insertion, which is time-consuming.

Multiset configuration

The simplest way to remove the duplicate assumption is to allow duplicates in the triplestore. This could be done by configuring the triplestore as a multiset instead of a normal set. A multiset is a set where multiple instances of the same element can occur. Unlike the counting duplicates approach, this solution does not use a different representation for duplicates. With a multiset, each occurrence of a triple corresponds to precisely one OTTR instance, allowing us to delete and insert triples safely. If we expand Listing 8.9 into a multiset, the result becomes Listing 8.10. If there are two equal triples, we can delete one without deleting the other. However, for some existing OTTR users, moving away from the RDF definition is not feasible, as it might not be compatible with other systems. Additionally, the triplestore might not support a multiset configuration.

```
ex:subClassOf[?child, ?parent] :: {
    ottr:Triple(?child, :subClass, ?parent),
    ottr:Triple(?child, :type, :Class)
}

ex:subClassOf(:A, :B) .
ex:subClassOf(:A, :C) .
```

Listing 8.9: subClass template

```
:A :type :Class .
:A :subClass :B .
:A :type :Class .
:A :subClass :C .
```

Listing 8.10: Expansion with multiset

8.3 Experimental evaluation

We will examine how the *Duplicate solution* compares to the baseline *Rebuild solution*. This includes how the two solutions measure in a typical use case, as well as finding the conditions where it is better to use the *Rebuild solution*. Lastly, we will see how the insert and delete operations compare.

Figure 8.2 illustrates how the *Duplicate solution* is faster than the *Rebuild solution* for most cases where the number of changes to duplicate instances is relatively low. Note that all changes in the tests insert or delete duplicate instances. According to Figure 8.3, using the *Duplicate solution* is no longer beneficial once the number of duplicate changes approaches 4000 in a triplestore with 20000 instances. From Figure 8.4, we see how the duplicate solution scales as the number of instances increases. However, Figure 8.5 shows how the *Rebuild solution* performs worse even though they both scale linearly with the number of instances.

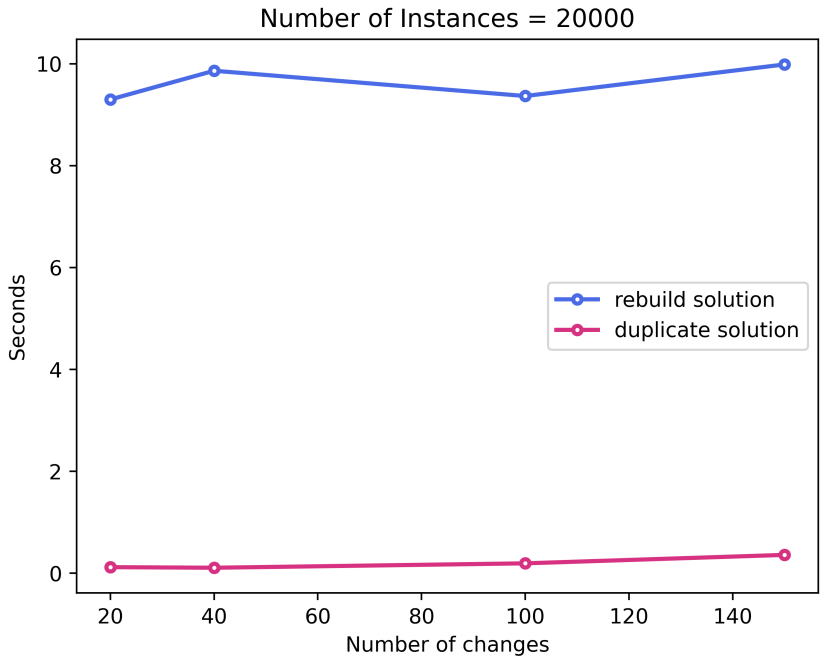

Figure 8.2: Duplicate solution VS. Rebuild solution

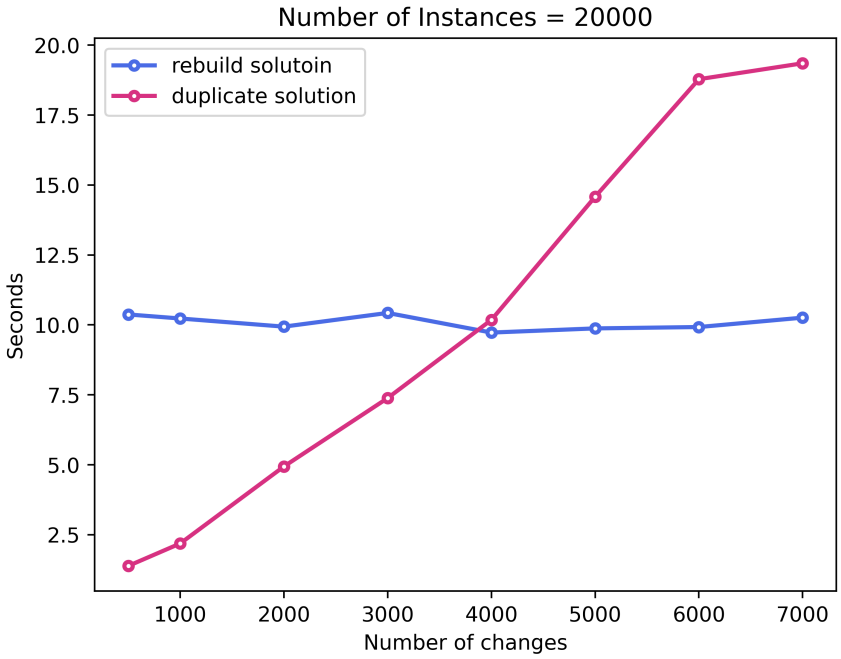

Figure 8.3: Duplicate VS. Rebuild point of intersection

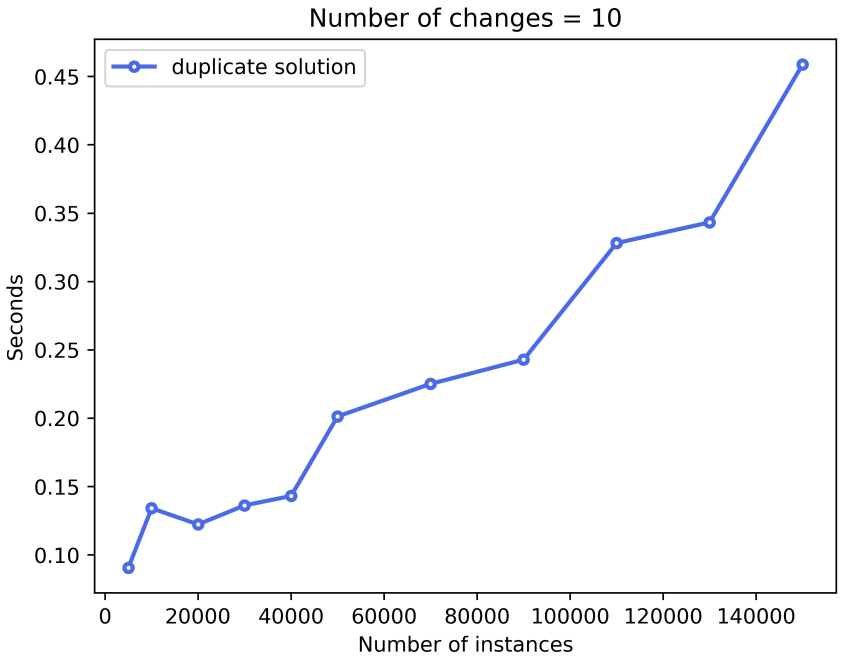

Figure 8.4: Varying number of instances

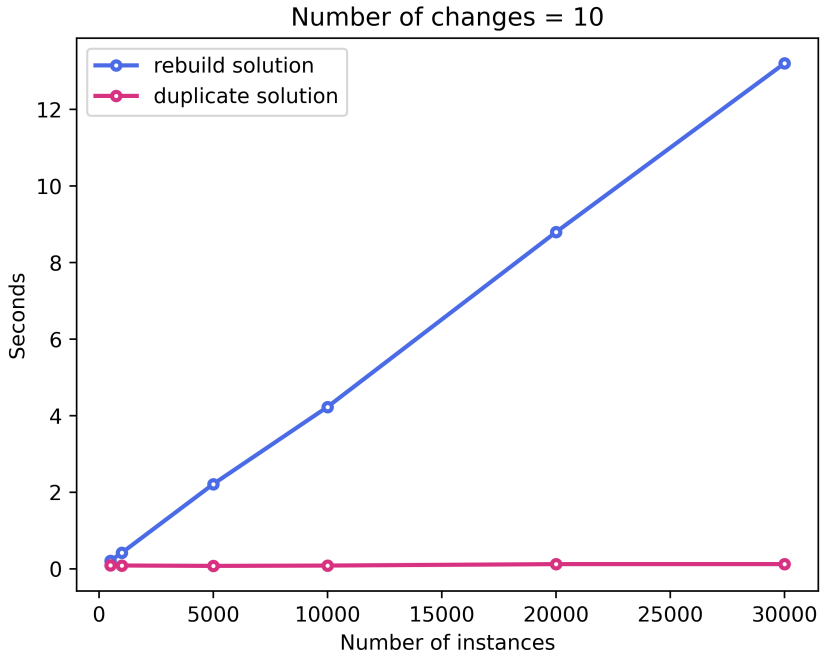

Figure 8.5: Duplicate VS. Rebuild changing number of instances

In Figure 8.6, we see how much time *Duplicate solution* uses when only inserting duplicate instances compared to only deleting duplicate instances. Both insertion and deletion scale linearly, and the difference in time between the two operations is minimal.

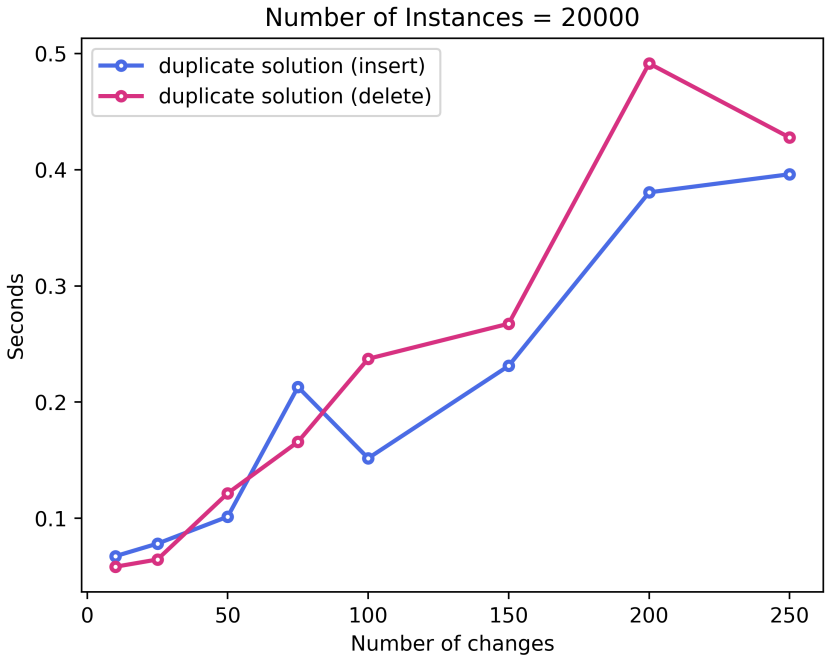

Figure 8.6: Duplicate solution, insertion vs. deletion

Figure 8.7 shows the *Duplicate solutions* when run on slightly different data. It compares how it performs on data with duplicate instances to how it performs normal instances. We see that it performs better when there are no duplicates.

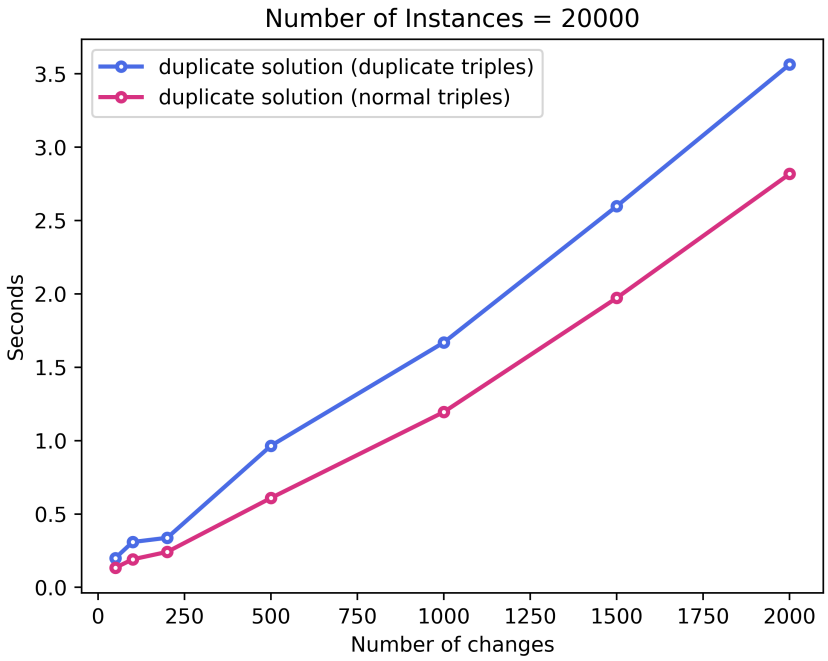

Figure 8.7: Duplicate solution, no duplicates vs. duplicates

8.4 Discussion

From the results presented in the previous section, we can see that the *Duplicate solution* significantly outperforms the *Rebuild solution* in cases where the number of changes does not make up a substantial portion of the total number of instances.

We know that the runtime of the *Rebuild solution* is independent of the number of changes. Therefore, it will take the same amount of time for the same number of instances, regardless of the number of changes. As *Duplicate solution* scales with the number of changes, we know that at a certain number of changes, using *Duplicate solution* is no longer advantageous. In Figure 8.3, this point is at about 4000 out of 20000 instances, or 20%. As discussed earlier in Section 5.1, typical use cases rarely change this many duplicates or this percentage of the OTTR instances. This example had 20000 instances, and we can expect the *Duplicate solution* to be even more efficient for a higher number of instances.

The key benefit of the *Duplicate solution* is enabling updates of instances that create or delete duplicates. The solution does this while still outperforming the *Rebuild solution*, as we see in Figure 8.5. Similar to the *Simple solution*, the *Duplicate solution* also scales linearly with the number of changes, as shown in Figure 8.3. It also scales linearly with the number of instances, shown in Figure 8.4, which the *Rebuild solution* also does. However,

the *Duplicate solution*'s linearly scaling operations are much faster than the *Rebuild solutions* linearly scaling operations, with respect to the total number of instances. Thus, we can conclude that the *Duplicate solution* performs better than the *Rebuild solution* as the difference between the number of changes and the size of the triplestore grows.

In contrast to the test results, a typical update will not change only duplicate instances. The *Duplicate solution* performs slightly better when updating non-duplicate instances than duplicate instances, as shown in Figure 8.7. This is because duplicate instances require extra operations to adjust the counter, which includes an HTTP request. However, this difference is small and will not be a decisive factor.

A downside to the *Duplicate solution* is that it has to check for duplicates in all operations. In a case where duplicates are rare but exist, we will always check for duplicates, but rarely find them. This might seem unnecessary, but it is essential for the *Duplicate solution* to work. This case will usually occur when there is a large triplestore with an empty TBox, which we assume to be typical for OTTR users.

Another drawback of this solution is the inclusion of RDF-star. The subject of the counter triple is the original triple represented in RDF-star. RDF* is not a part of the current RDF standard, but it is a popular extension and is currently supported by multiple RDF graph database systems like Apache Jena Fuseki [12], Blazegraph [30], and OpenDB [8]. Furthermore, an RDF-star working group has been created to extend a set of RDF and SPARQL-related recommendations, with the ability to represent and query statements about statements [41]. This means that RDF-star or something similar will likely be part of the RDF standard in the near future. As RDF* is commonly supported, and we argue that using it in this solution is appropriate.

Lastly, while this solution removes the assumption that duplicates cannot be deleted, it still cannot handle duplicate statements containing blank nodes or blank nodes in general. This will be addressed in the following chapters.

Chapter 9

Removing the local blank node assumption

This chapter will start by illustrating the problems that occur without the local blank node assumption. We will discuss the identified problems and present solutions. Then, we will look at the implementation of the *Blank node solution*. Lastly, we will show and discuss the performance of this solution.

Blank nodes, which allow users to denote resources without an IRI or other value, are an important and valuable part of RDF. However, they present a challenge when developing an update algorithm. Blank nodes can only be referenced by their relations to other resources as they do not have an IRI, and their label cannot be used as they are local in scope. The *local blank node assumption* is the assumption that triples containing blank nodes cannot be deleted. In the following sections, we will explore how this assumption can be loosened and what considerations must be taken into account. We will retain the assumption that the OTTR instances in the instance file (*top-level instances*) are not given blank nodes as arguments, as we discussed in Section 5.1. We will still assume that duplicates do not exist to focus on the blank node problem in isolation. The issue of removing both the local blank node assumption and the duplicate assumption simultaneously will be addressed in Chapter 10.

The challenge with local blank nodes can be illustrated by the examples in Listing 9.1 through Listing 9.4. Listing 9.1 and Listing 9.2 show the OTTR files and the corresponding triplestore before the update. Bob has a child Kari, Kari has a child Lisa, and Bob is the grandparent of Lisa. If we want to delete `ex:HasGrandchild(<bob>, <lisa>)` and follow the rules of the *Simple solution*, we get the query in Listing 9.3. Note that we have replaced the blank nodes with variables in the query, as they act similarly. We want to delete two triples from the triplestore, `(<bob> :hasChild _:b1)` and `(_:b1 :hasChild <lisa>)`, but with this `DELETE`

query we will delete all four triples, shown in Listing 9.4. This is because the triples from `ex:HasChild(<bob>, <kari>)` and `ex:HasChild(<kari>, <lisa>)` together form the same pattern. Templates can be significantly more complex than in the previous example. For example, a template may contain multiple non-base templates instances with the blank node as an argument. This increases the size of the matching graph. The complexity also increases when there are multiple different local blank nodes.

```
ex:HasChild[?person, ?child] :: {
    ottr:Triple(?person, :hasChild, ?child)
}

ex:HasGrandchild[?person, ?grandchild] :: {
    ex:HasChild(?person, _:blankChild),
    ex:HasChild(_:blankChild, ?grandchild)
}

ex:HasChild(<bob>, <kari>) .
ex:HasChild(<kari>, <lisa>) .
ex:HasGrandchild(<bob>, <lisa>) . #deleting this
```

Listing 9.1: Two instances creating a blank node

```
# should not be deleted
<bob> :hasChild <kari> .
<kari> :hasChild <lisa> .

# should be deleted
<bob> :hasChild _:b1 .
_:b1 :hasChild <lisa> .
```

Listing 9.2: Expanded graph

```
DELETE WHERE {
    <bob> :hasChild ?blankChild .
    ?blankChild :hasChild <lisa> .
}
```

Listing 9.3: Delete query

```
# should not be deleted
<bob> :hasChild <kari> .
<kari> :hasChild <lisa> .

# should be deleted
<bob> :hasChild _:b1 .
_:b1 :hasChild <lisa> .
```

Listing 9.4: Resulting graph, too much has been deleted

9.1 Problems

The main issue with blank nodes is making a query describing exactly the triples we want and no others. If we use the pattern from an expanded instance with blank nodes as variables, then the variables can match other unintended resources in addition to the blank nodes. In this section, we will identify the different cases where we can get undesirable results and how to solve those problems. The problems will only occur when deleting triples, as we will struggle to identify the triples to delete. Insertion works the same way as in the *Simple solution* and does not pose any problem. The result of expanding the instances to delete will be the pattern we are searching for, referred to as *the desired pattern*.

There are three cases where the *Simple solution* can give us a wrong result, explained in more detail below:

1. An non-blank resource creates *the desired pattern* across multiple top-level instances
2. There exists an equivalent graph matching the pattern
3. There is a template creating a super-set of *the desired pattern*

Non-blank resource creates the desired pattern

We know blank nodes cannot be passed as arguments to top-level instances. However, identifiable resources (IRIs and literals) can be top-level arguments, meaning they can form the desired pattern across the expansion of several instances. This is what happened in Listing 9.1, Listing 9.3 and Listing 9.4. We are looking for the blank node that is the child of Bob and the parent of Lisa, however, the resource `<kari>` matches that description. This can be solved by ensuring that the variables are blank nodes. This is because a blank node only exists within the expansion of a single OTTR instance, and therefore cannot form the desired pattern across the expansion of several instances.

An equivalent graph

If expanding another instance results in an equivalent graph to the desired pattern, we will match too many triples. In Listing 9.5 and Listing 9.6, there is an example of two different instances that expands to equivalent patterns. If we try to delete `?person :hasName "Clark"`, we will delete both the result of person and employee, which is incorrect. We have no way of differentiating the person and the employee, which means it does not matter which one is deleted, but it is important to delete only one of them. If we can ensure that we only delete the expansion of one of the equivalent instances, then this will not be a problem. Note that this only applies when two instances create the exact same pattern without additional triples for one of them. We can solve this by limiting the number of times a deletion operation is allowed to run, regardless of the number of matches.

```
ex:Person[?name] :: {
    ottr:Triple(_:person, :hasName, ?name)
}

# company is optional
ex:Employee[?name, ? ?company] :: {
    ottr:Triple(_:person, :hasName, ?name),
    ottr:Triple(_:person, :worksAt, ?company)
}

ex:Person("Clark") .
ex:Employee("Clark", none) .
```

Listing 9.5: Creating equivalent graphs

```
# result from person
_:b1 :hasName "Clark" .

# result from employee
_:b2 :hasName "Clark" .
```

Listing 9.6: Expanded graph, equivalent sets

Template creating a super-set of the desired pattern

If we want to delete a set of triples $\mathcal{A}$, and $\mathcal{A}$ is a subset of $\mathcal{B}$, $\mathcal{A} \subset \mathcal{B}$, then the *Simple solution* will delete both $\mathcal{A}$ and that subset of $\mathcal{B}$. For example, in Listing 9.7, there are two templates, `ex:Person` and `ex:Superperson`. The `ex:Superperson` creates exactly the same triple as `ex:Person` in addition to a triple about the superpower. We also have two instances, `ex:Person("Clark")` and `ex:Superperson("Clark", "flying")`. Both put

"Clark" as name, which means that two equivalent triples will be made as seen in Listing 9.8. Note that these two triples are not identical as there will be different blank nodes in them, but since blank nodes are anonymous, we cannot differentiate them and call them equivalent. Even though both triples are equivalent, we cannot delete a random one as one belongs to the Superperson. If `(_:b2 :hasName "Clark")` is deleted, then the flying person has no name, and there is still another person named Clark.

To solve this problem, we need to ensure that we do not match with the result of templates that create a super-set of the desired pattern. We know that blank nodes cannot exist across top-level instances as they cannot take blank nodes as arguments, meaning all occurrences of a specific blank node is from the same top-level instance. We also know that we cannot match with non-blank resources due to the solution to our first problem. If we ask for the desired pattern, we can get multiple blank nodes as a result. If we then count the occurrences of those blank nodes, we can see if a blank node exists in additional triples. If it does, it cannot be the blank node we seek, as it must be from a different instance.

For example, if we delete the person instance in Listing 9.7 (line 1 in Listing 9.8), we know that the blank node exists in exactly one triple. We can then use the pattern `?person :hasName "Clark"` to find the blank nodes matching. We then get two blank nodes, `_:b1` and `_:b2`, but we count that `_:b2` occurs twice, meaning it cannot be from the deleted person. If several blank nodes match the pattern and no one occurs in additional triples, it does not matter which one we delete, as shown in the previously discussed problem (An equivalent graph). We can then delete a random one, as it does not matter which one we delete.

```
1  ex:Person[?name] :: {
2      ottr:Triple(_:person, :hasName, ?name)
3  }
4
5  ex:Superperson[?name, ?superpower] :: {
6      ottr:Triple(_:person, :hasName, ?name),
7      ottr:Triple(_:person, :hasSuperpower, ?superpower)
8  }
9
10 ex:Person("Clark") . #deleting this
11 ex:Superperson("Clark", "flying") .
```

Listing 9.7: Template creating a super set

```
# person
_:b1 :hasName "Clark" . # equivalent triple

# superperson
_:b2 :hasName "Clark" . # equivalent triple
_:b2 :hasSuperpower "flying" .
```

Listing 9.8: Expanded graph

9.2 Implementation

The *Blank node solution* uses the same approach as the *Simple solution*. A single module that uses Jena model, Apache Jena Updatebuilder, OTTR-service, Fuseki-service, and Diff-service. The difference between the *Blank node solution* and the *Simple solution* is that the *Blank node solution* has to identify all blank nodes in the update, and create a more complex SPARQL query when deleting. This section will look at the case where one instance, which creates a single blank node, is deleted. Insertion in the *Blank node solution* is the same as the *Simple solution*, so we will only show deletion. We use the previously explored example from Listing 9.1, and we delete the `ex:HasGrandChild(<bob>, <lisa>)` instance. The result of this section is a complicated query consisting of several parts. We will walk through these parts one at a time and build the complicated query. The entire query can be seen in Listing 9.15. We use the method from *Simple solution* as the starting point for our query: Listing 9.9. Then, we will adapt the solutions to the three problems to build our query. An illustrated overview of the *Blank node solution* can be found in Figure 9.1.

```
DELETE WHERE {
    <bob> :hasChild ?blankChild .
    ?blankChild :hasChild <lisa> .
}
```

Listing 9.9: Delete query, simple solution

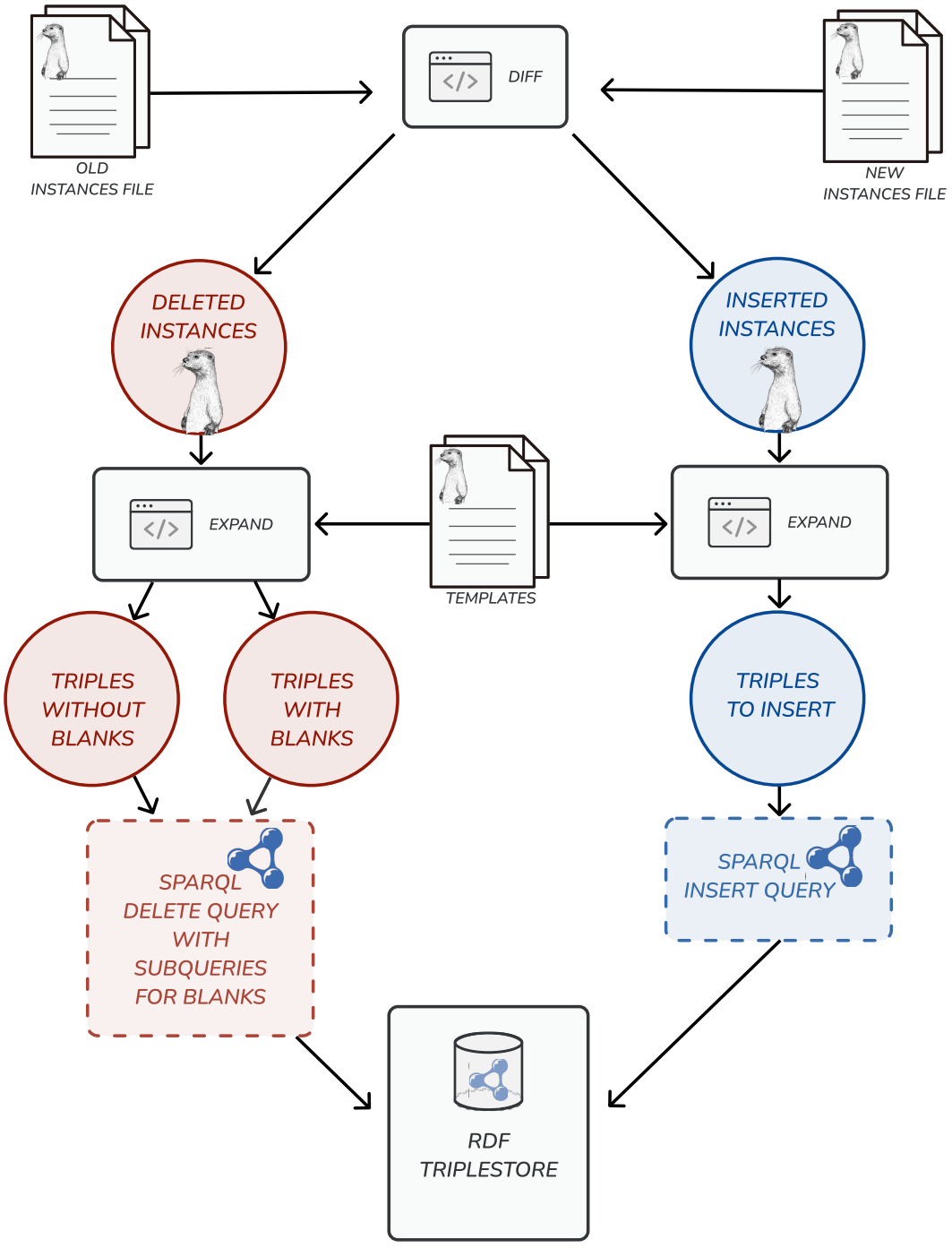

Figure 9.1: Blank node solution overview

Non-blank resource creates pattern

We have to ensure that the variables represent blank nodes. This can be done with the `isBlank()` function. `isBlank(?variable)` returns true if `?variable` is a blank node. By adding `FILTER(isBlank(?variable))` to our query, we only get triples where `?variable` is a blank node. The query is now updated to Listing 9.10.

```
DELETE {
    <bob> :hasChild ?blankChild .
    ?blankChild :hasChild <lisa> .
}
WHERE {
  <bob> :hasChild ?blankChild .
  ?blankChild :hasChild <lisa> .
  FILTER (isBlank(?blankChild))
}
```

Listing 9.10: SPARQL query with isBlank(). ?blankChild must be a blank node

An equivalent graph

We now know it does not matter which graph we delete as long as the pattern is the entire expansion of the instance and we only delete one of the graphs. In two equivalent graphs, all the normal triples are the same, and the different blank nodes relate to the same resources. Deleting one or the other yields the same result. In SPARQL, only using the desired pattern can match several equivalent graphs, but we can use `LIMIT 1` to return only one. Our query now looks like Listing 9.11.

```
DELETE {
    <bob> :hasChild ?blankChild .
    ?blankChild :hasChild <lisa> .
}
WHERE {
  <bob> :hasChild ?blankChild .
  ?blankChild :hasChild <lisa> .
  FILTER (isBlank(?blankChild))
  LIMIT 1
}
```

Listing 9.11: Delete query, second attempt with LIMIT

Template creating a super set of the pattern

This is a more complicated problem that will have a complicated query as a solution. The solution to the problem can be divided into three parts:

1. Find all blank nodes that match the pattern
2. Count the number of occurrences of each of these blank nodes
3. Delete a single set of triples where all blank nodes occur the expected number of times

The structure of the query is described in Listing 9.12.

```
DELETE {
    # 3. DESIRED PATTERN
    <PATTERN>
}
WHERE {
    {
    # 2. COUNT OCCURRENCES
    SELECT ... AS ?num_blank)
    WHERE {
        # 2. BLANK NODE CAN BE SUBJECT OR OBJECT
        <COUNTING_PATTERN>

        # 1. FIND BLANK NODES MATCHING THE PATTERN
        {
            SELECT ?blank ...
        }
    }
    # 3. RETURN ONE BLANK NODE WITH EXPECTED COUNT
    ...
    }
}
```

Listing 9.12: Structure of the blank node delete query

To solve this problem, we have to use the solutions to the other problems. First, we must find all blank nodes that match the pattern. This is already done with `FILTER(isBlank(?variable))` in Listing 9.10. We update the example query: Listing 9.13.

```
DELETE {
    # 3. DESIRED PATTERN
    <PATTERN>
}
WHERE {
    {
    # 2. COUNT OCCURRENCES
    SELECT ... AS ?num_blank)
    WHERE {
        # 2. BLANK NODE CAN BE SUBJECT OR OBJECT
        <COUNTING_PATTERN>

        1. FIND BLANK NODES MATCHING THE PATTERN
        {
            SELECT ?blankChild
            WHERE {
                <bob> :hasChild ?blankChild .
                ?blankChild :hasChild <lisa> .
                FILTER (isBlank(?blankChild)) .
            }
        }
    }
    # 3. RETURN ONE BLANK NODE WITH EXPECTED COUNT
    ...
    }
}
```

Listing 9.13: Blank node delete query. Finding the blank nodes.

Now we have found the blank nodes and must count the occurrences of each one. We can do this by using the `COUNT` aggregate function and grouping the blank nodes. To find this set, we have to provide a pattern to search for. Blank nodes can only occur in the subject and object positions of a triple, and therefore we must search for all triples where one of our blank nodes is in the subject or object position. `(?blank ?pred ?obj)` matches the subject position, and `(?sub ?pred ?blank)` matches the object position. We add these results together by using `UNION`. We update the query: Listing 9.14.

```
DELETE {
    # 3. DESIRED PATTERN
    <PATTERN>
}
WHERE {
    {
    2.  COUNT OCCURRENCES
    SELECT ?blankChild (COUNT(?blankChild) AS ?num_blankChild)
    WHERE {
        2.  BLANK NODE CAN BE SUBJECT OR OBJECT
        {?blankChild ?pred ?obj}
        UNION
        {?sub ?pred ?blankChild}

        # 1. FIND BLANK NODES MATCHING THE PATTERN
        {
            SELECT ?blankChild
            WHERE {
                <bob> :hasChild ?blankChild .
                ?blankChild :hasChild <lisa> .
                FILTER (isBlank(?blankChild))
            }
        }
    }
    GROUP BY ?blankChild

    # 3. RETURN ONE BLANK NODE WITH EXPECTED COUNT
    ...
    }
}
```

Listing 9.14: Blank node delete query. Counting the blank nodes.

Lastly, we have to find which blank nodes occur the correct number of times in the triplestore and then delete the pattern with one of those blank nodes. We have already counted the occurrences of the blank nodes that match the desired pattern by grouping the blank nodes. Now we can use `HAVING` to check whether the count is as expected. In the example, the expected count is two since the blank node occurs in two triples when expanding the deleted instance; see Listing 9.2. `LIMIT 1` can be used to return only one blank node if several have the correct count. Then we put the desired pattern in the `DELETE` clause. The query will now work in our example, see Listing 9.15, but we have to adjust it if there are multiple different blank nodes in the expansion of the deleted instances.

```
DELETE {
    3.  PATTERN WE ARE SEARCHING FOR
    <bob> :hasChild ?blankChild .
    ?blankChild :hasChild <lisa> .
}
WHERE {
    {
    # 2. COUNT OCCURRENCES
    SELECT ?blankChild (COUNT(?blankChild) AS
       ?num_blankChild)
    WHERE {
    # 2. BLANK NODE CAN BE SUBJECT OR OBJECT
        {?blankChild ?pred ?obj}
        UNION
        {?sub ?pred ?blankChild}

        # 1. FIND BLANK NODES MATCHING THE PATTERN
        {
            #FIND BLANK NODES
            SELECT ?blankChild
            WHERE {
                <bob> :hasChild ?blankChild .
                ?blankChild :hasChild <lisa> .
                FILTER (isBlank(?blankChild))
            }
        }
    }
    GROUP BY ?blankChild

    3.  RETURN ONE BLANK NODE WITH EXPECTED COUNT
    HAVING (?num_blankChild = 2)
    LIMIT 1
    }
}
```

Listing 9.15: Blank node delete query. Delete one correct set of triples.

9.3 Multiple blank nodes

Having multiple blank nodes in the same pattern increases the complexity of the implementation in some parts. Although most of the considerations remain unchanged, a few changes must be implemented. Each blank node is given its own variable in the SPARQL query. In addition, we must ensure that the match is exact for all variables, which means we need a subquery for each blank node. The subquery consists of everything we have seen so far inside the outer `WHERE` clause. If we have two blank nodes in the pattern, the query would look like Listing 9.16.

This solution will work most of the time, but there is a complicated special case if two of the same blank node are in a single triple, `(_:b :hasChild _:b)`. `_:b` will get count two from this single triple, as it appears in the subject and object position. This single triple can also be used to match several triples when searching for the desired pattern. If the triplestore looks like Listing 9.17, and we want to delete the expansion of the second instance, then we are creating a subquery for `_:b1`, `_:b2` and `_:b3` as previously discussed. The problem occurs in the subquery for `_:b2`, as shown in Listing 9.18. First, `_:b` and `_:b2` will be returned as a result of the inner subquery, `_:b` is returned because `(_:b :hasChild _:b)` matches both triples in the pattern. Then `(_:b :hasChild _:b)` will occur twice when taking the union of having the blank node in the subject position and the object position, as `_:b` occurs in both positions. When we count the number of triples in which the blank nodes occur, `(_:b :hasChild _:b)` is counted twice, and `_:b` gets the same count as `_:b2`, which means that we will delete a random of them for `?b2` when deleting. If `_:b` is chosen, then nothing will be deleted since `(_:b1 :hasChild _:b)` and `(_:b :hasChild _:b3)` do not exist in the triplestore. We can easily solve this by removing duplicates when counting triples. This can be done with the `DISTINCT` keyword. The implementation of the solution is now done, and the query will look like Listing 9.19.

```
DELETE {EXPANSION OF ALL DELETED INSTANCES:
    <id1> :p ?blank1 .
    ?blank1 :p ?blank2 .
    <id2> :p ?blank2 .
    <id3> :p ?blank2 .
} WHERE {
    {SUBQUERY FOR BLANK1
    SELECT ?blank1 (COUNT(?blank1) AS ?num_blank1)
    WHERE {
        {?blank1 ?pred ?obj}
        UNION
        {?sub ?pred ?blank1}
        {
            SELECT ?blank1
            WHERE {
                <id1> :p ?blank1 .
                ?blank1 :p ?blank2 .
                FILTER (isBlank(?blank1))
            }
        }
    }
    GROUP BY ?blank1
    HAVING (?num_blank1 = 2)
    LIMIT 1
    }
    {SUBQUERY FOR BLANK2
    SELECT ?blank2 (COUNT(?blank2) AS ?num_blank2)
    WHERE {
        {?blank2 ?pred ?obj}
        UNION
        {?sub ?pred ?blank2}
        {
            SELECT ?blank2
            WHERE {
                ?blank1 :p ?blank2 .
                <id2> :p ?blank2 .
                <id3> :p ?blank2 .
                FILTER (isBlank(?blank2))
            }
        }
    }
    GROUP BY ?blank2
    HAVING (?num_blank2 = 3)
    LIMIT 1
    }}
```

Listing 9.16: Delete query with two blank nodes

```
# expansion of one instance
_:b :hasChild _:b .

# expansion of another instance, will be deleted
_:b1 :hasChild _:b2 .
_:b2 :hasChild _:b3 .
```

Listing 9.17: triplestore, special case

```
DELETE {EXPANSION OF ALL DELETED INSTANCES:
    ?b1 :hasChild ?b2  .
    ?b2 :hasChild ?b3 .
} WHERE {
    SUBQUERY FOR b1 AND b3 analogous to SUBQUERY FOR b2
    {SELECT ?b1 ...} {SELECT ?b3 ...}

    {SUBQUERY FOR b2
    SELECT ?b2 (COUNT(?b2) AS ?num_b2)
    WHERE {
        {?b2 ?pred ?obj}
        UNION
        {?sub ?pred ?b2}
        {
            SELECT ?b2
            WHERE {
                ?b1 :hasChild ?b2 .
                ?b2 :hasChild ?b3 .
                FILTER (isBlank(?b2))
            }
        }
    }
    GROUP BY ?b2
    HAVING (?num_b2 = 2)
    LIMIT 1
    }}
```

Listing 9.18: Subquery matching too many blank nodes

```
DELETE {EXPANSION OF ALL DELETED INSTANCES:
    ?b1 :hasChild ?b2 .
    ?b2 :hasChild ?b3 .
} WHERE {
    SUBQUERY FOR b1 AND b3
    {SELECT ?b1 ...} {SELECT ?b3 ...}

    {SUBQUERY FOR b2
    SELECT ?b2 (COUNT(DISTINCT *) AS ?num_b2)
    WHERE {
        {?b2 ?pred ?obj}
        UNION
        {?sub ?pred ?b2}
        {
            SELECT ?b2
            WHERE {
                ?b1 :hasChild ?b2 .
                ?b2 :hasChild ?b3 .
                FILTER (isBlank(?b2))
            }
        }
    }
    GROUP BY ?b2
    HAVING (?num_b2 = 2)
    LIMIT 1
    }}
```

Listing 9.19: Complete delete query

9.4 Experimental evaluation

The following section shows the performance of the *Blank node solution* with a varying number of operations on blank instances. First, we compare the *Blank node solution* to the *Rebuild solution* in a typical use case and then find the point where rebuilding outperforms the *Blank node solution*. In addition, we will compare the insert and delete operations.

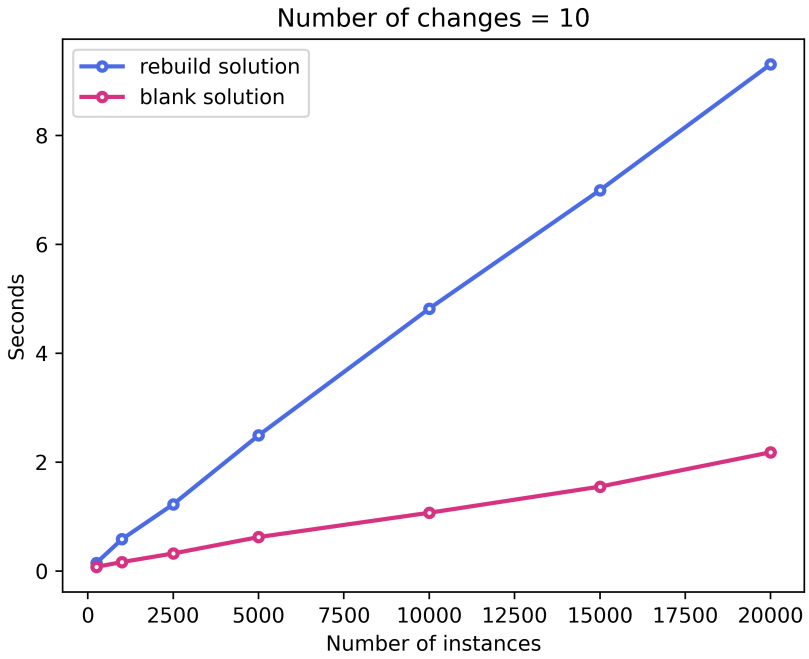

Figure 9.2: Blank node solution VS. Rebuild solution, varying instances

Figure 9.2 illustrates how the *Blank node solution* compares to the *Rebuild solution*. We observe that both solutions scale linearly with the number of instances, but the linear scaling operation in the *Blank node solution* is faster than the linear scaling operation in the *Rebuild solution*. In Figure 9.3, we see that for low numbers of changes to blank nodes, the *Blank node solution* is faster. However, once the number of changes is around 60, the *Rebuild solution* is more efficient. While the *Rebuild solution* uses the same time regardless of the number of changes, we see that the *Blank node solution* run time grows linearly with the number of changes. Another interesting observation is in Figure 9.4 where the number of blank deletions and insertions is a constant of five, respectively. We see that the insert operation is not particularly affected by the number of instances, unlike the delete operation, which scales linearly with the number of instances.

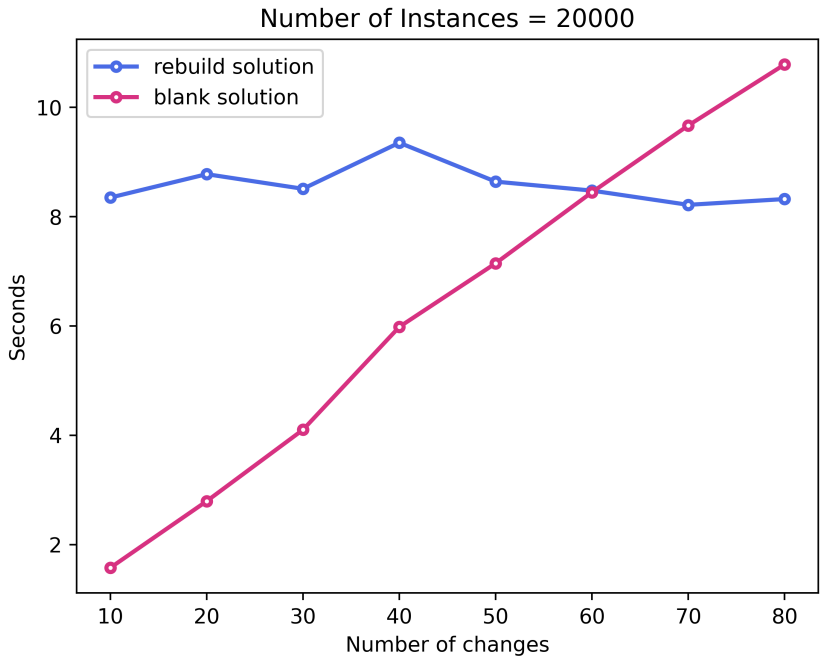

Figure 9.3: Blank node solution VS. Rebuild solution, varying changes

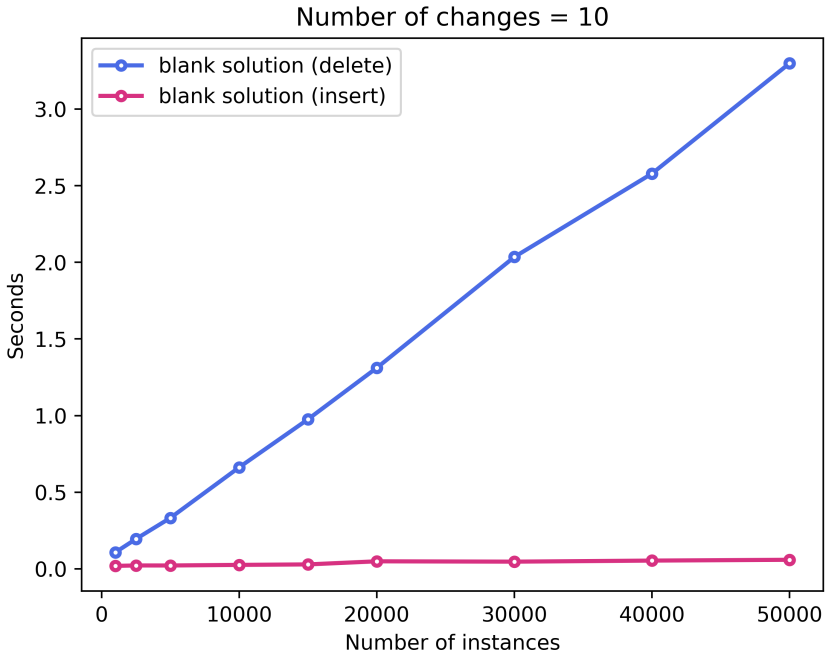

Figure 9.4: Deleting only blank nodes vs. inserting only blank nodes

Insertion also scales better than deletion as the number of changes increases. In Figure 9.5, we compare the *Blank node solution* performing only insert operations, only delete operations, and the *Rebuild solution*. We can see that the *Blank node solution* is quicker at inserting blank nodes than deleting blank nodes. We see in Figure 9.6 that the point of intersection for inserting blank nodes is about 12500 out of 20000. This happened already at 25 for deletion, as shown in Figure 9.5. Figure 9.7 shows that when there are changes to blank nodes, the query execution part of the algorithm is by far the most time-consuming.

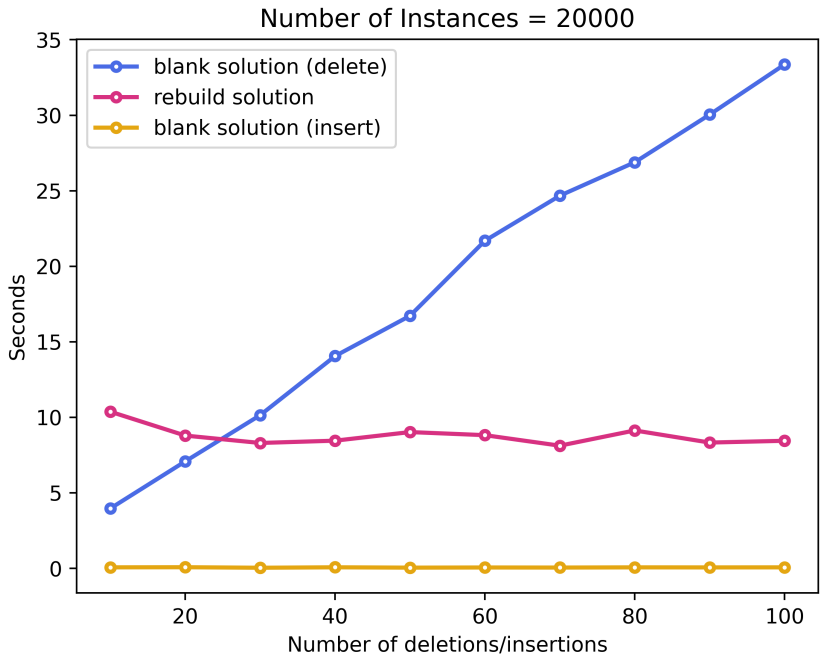

Figure 9.5: Inserting blank nodes vs. deleting blank nodes vs. rebuilding, point of intersection delete

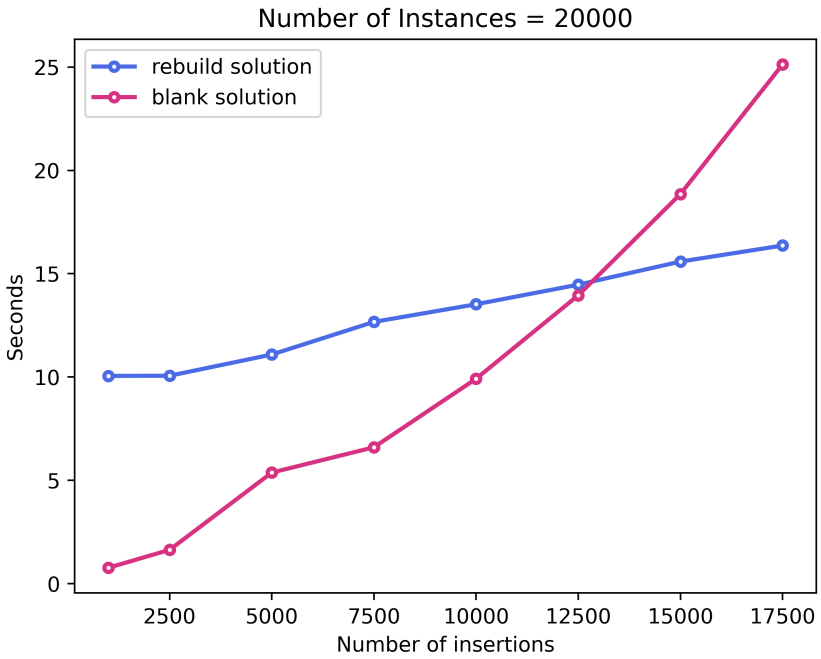

Figure 9.6: Inserting blank nodes vs. rebuilding, point of intersection insert

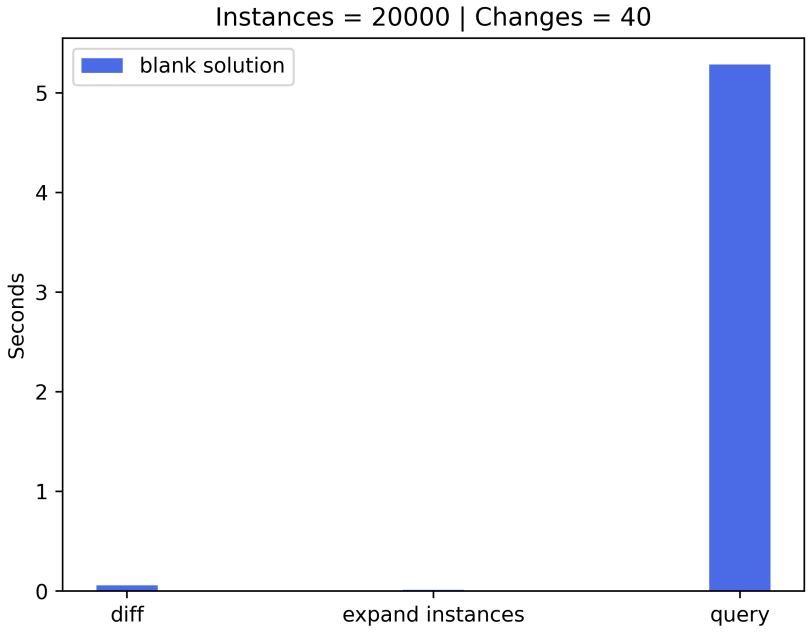

Figure 9.7: Time spent on different parts of the algorithm

9.5 Discussion

One of the key advantages of the *Blank node solution* is its ability to remove the assumption that blank nodes cannot be deleted. Moreover, the *Blank node solution* generally outperforms the *Rebuild solution* in the typical case from Section 5.1, as we see in Figure 9.2. However, the algorithm performs much worse when deleting blank nodes than when inserting blank nodes. The point where this solution is no longer viable is highly dependent on how many blank nodes that are deleted, as we see in Figure 9.5 and Figure 9.6. In a graph with 20000 instances, rebuilding is better already when deleting 25 blank nodes, while the same point for insertion is at about 12500 blank nodes. The significant discrepancy is the result of blank node insertion being done similarly to the *Simple solution*, while deletion of blank nodes requires pattern matching in the triplestore. In this test, we see that deleting a blank instance is approximately 500 times slower than inserting a blank instance.

Another advantage of the *Blank node solution* is that non-blank instances are treated in the same way as in the *Simple solution*. The only difference is a check whether there exist blank nodes in the updated triples, which will not be noticeable. This trait means that we can use the *Blank node solution* instead of the *Simple solution* in all cases without losing much performance. Later in Chapter 11, we will see that the *Blank node solution* will match the performance of the *Simple solution* as long as blank nodes are absent.

As we know, the *Rebuild solution* scales constantly with the number of changes. The *Blank node solution*, on the other hand, scales both with the number of changes and the number of instances. This means that at a certain number of changes, the *Blank node solution* will no longer be more efficient than the *Rebuild solution*. As we discussed in Section 5.1, we assume most data have few blank nodes, meaning we can assume that most realistic updates do not change enough blank nodes for the *Rebuild solution* to outperform the *Blank node solution*.

Chapter 10

Combined solution

The natural continuation and conclusion of the three previous chapters is a combination of all three solutions. By creating a solution with both the duplicate assumption and the local blank node assumption removed, we end up with a fairly complete solution. We call this solution the *Combined solution*. The only remaining assumption is that no blank node is passed as an argument to a top-level instance. This chapter will present an outline for the *Combined solution*. Following this, the results from the experimental evaluation are presented. Lastly, we discuss the results.

10.1 Description

Implementing the *Combined solution* will require only minor adaptations to the other solutions. This section will outline what adaptations must be done to combine the different solutions. We use the methods from the *Blank node solution* to handle blank triples, and use the methods from the *Duplicate solution* to handle all other triples. Duplicates of blank triples can be ignored in this solution, which we will explain later. Thus, the methods from *Duplicate solution* and *Blank node solution* handle separate input data and will not interfere.

First, we expand the updated instances and split the triples into two categories. Triples with blank nodes and triples without. Deletion and insertion of non-blank triples will work as described in the *Duplicate solution*. Deleting and inserting blank triples will work as in the *Blank node solution*. As this solution keeps the assumption that no blank node is passed as a top-level argument, it follows that a single blank node can only originate from one single OTTR instance, meaning that two distinct instances cannot create triples with the same blank node. Additionally, we know from Section 8.1 that duplicate triples originating from the same instance can be ignored. We now know that duplicated blank triples only

can occur from the same instance, and we also know that we can ignore duplicates from the same instance. Thus, we can ignore duplicated blank triples in general. Therefore, we can use the operations from the *Blank node solution* to handle blank triples, instead of combining it with *Duplicate solution*.

The *Combined solution* consists of three steps. First, the algorithm separates the triples containing blank nodes from the ones not containing blank nodes. Secondly, the blank triples are handled. Lastly, we handle the non-blank triples. We will not present the insert and delete operations implemented in pseudo-code:

INSERT

in the algorithm below, we let:
hm=counting hash map, *bs*=blank set, *i* = instance, *t* = triple

1. FOR each instance *i* to be inserted
    (a) FOR each triple *t* in expansion of *i*
        i. IF *t* contains blank nodes, THEN
            A. Add *t* to *bs*
        ii. ELSE
            A. Add *t* to *hm* and increment its count
2. FOR each triple *t* in *bs*
    (a) Insert *t* using the blank node solution
3. FOR each triple *t* in *hm*
    (a) Insert *t* using the duplicate solution

DELETE

in the algorithm below, we let:
hm=counting hash map, *bs*=blank set, *i* = instance, *t* = triple

1. FOR each instance *i* to be deleted
    (a) FOR each triple *t* in expansion of *i*
        i. IF *t* does not contain blank nodes, THEN
            A. add *t* to *hm* and increment its count
        ii. ELSE
            A. Add *t* to *bs*
2. FOR each triple *t* in *bs*
    (a) delete *t* with the blank node algorithm
3. FOR each *t* in *hm*
    (a) Delete *t* using the duplicate algorithm

The *Combined solution* is implemented using a similar approach as the previous solutions. It is its own module that interacts with the OTTR-service, Diff-service, and other services. Additionally, this module imports the *Duplicate solution* and the *Blank node solution* modules in order to have access to their functions. What sets this implementation apart from the previous solutions is the fact that the triples are separated into categories initially; a Model containing blank triples, and a HashMap containing normal triples. Figure 10.1 illustrates an outline of the combined solution.

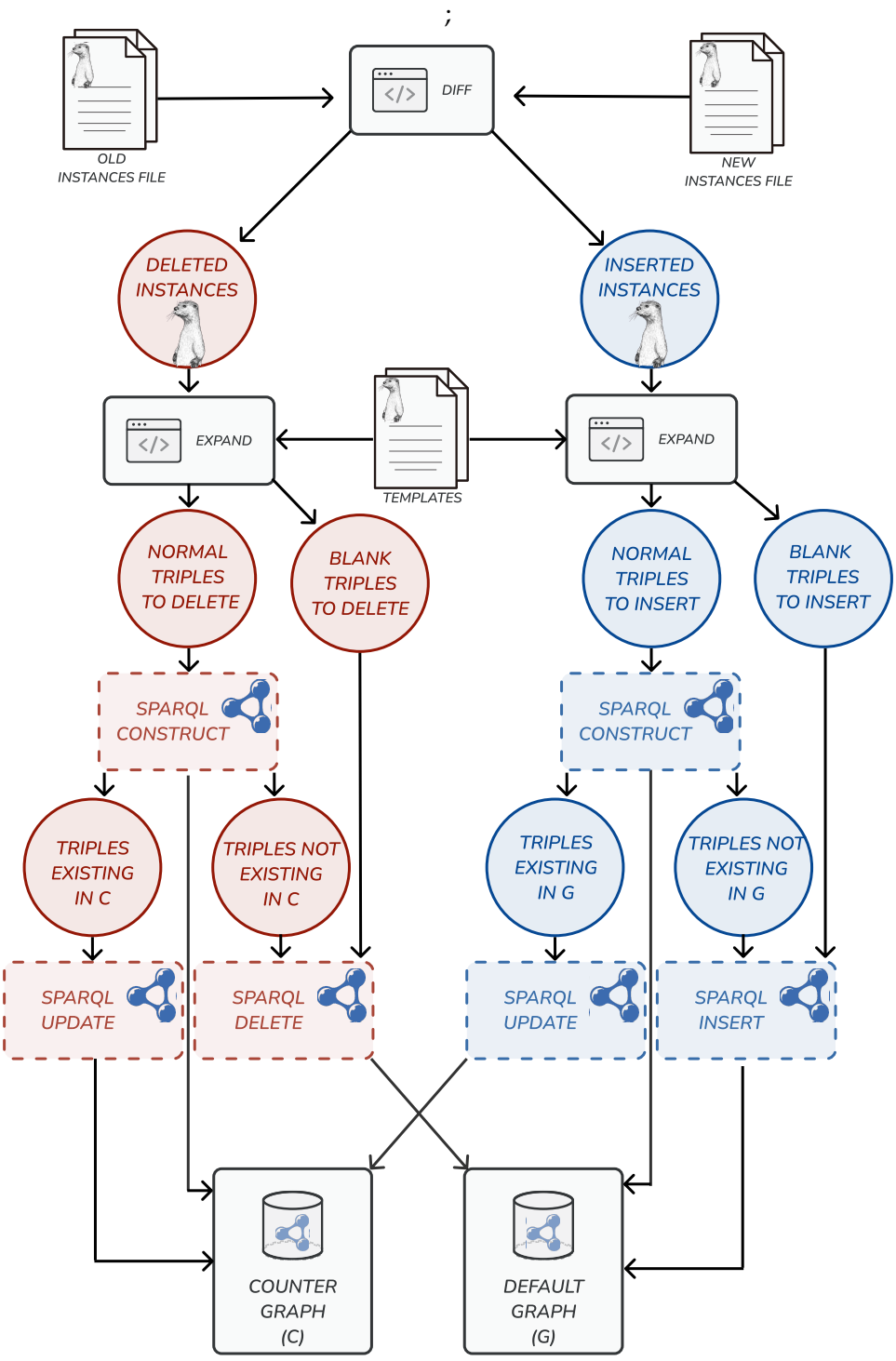

Figure 10.1: Outline combined solution

10.2 Experimental evaluation

This section presents the results of our experiments. As the *Combined solution* handles blank tiples and other triples separately, we expect the *Combined solution* to perform similarly to the *Blank node solution* and the *Duplicate solution* at their respective tasks.

In Figure 10.2, we see how the *Combined solution* and *Duplicate solution* performs equally in a case with 20000 instances, and all changes are made to duplicate instances. The figure in Figure 10.3 illustrates another test case with a similar number of instances and a varying number of changes to only blank instances. Here we see the *Combined solution* performs similarly to the *Blank solution*.

Figure 10.4 gives an example of how the *Combined solution* performs in a case with all types of changes and a triplestore of size 10000. We insert and delete 5 separate blank instances, duplicate instances, and normal instances. We see that the *Combined solution* scales better than the *Rebuild solution*.

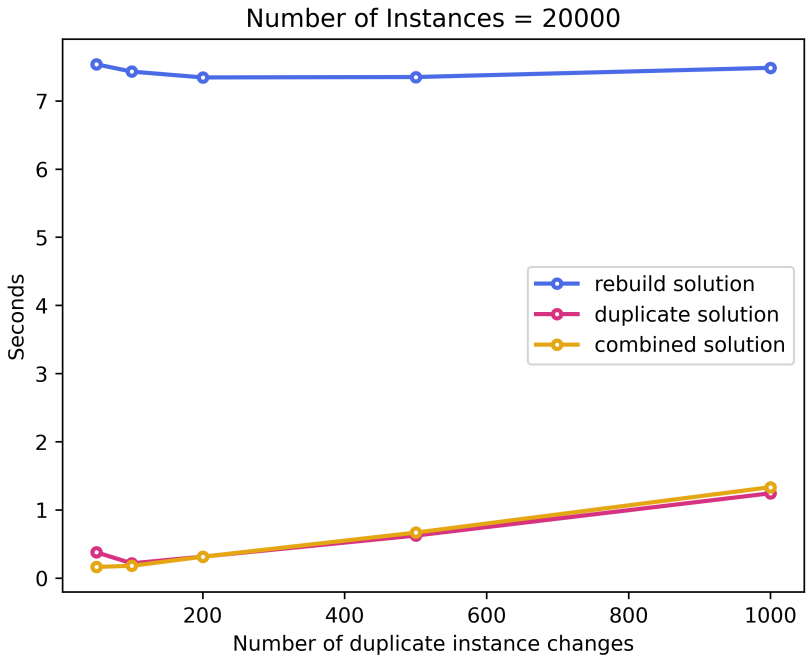

Figure 10.2: Combined, Duplicate, and Rebuild solution, varying number of duplicate instance changes

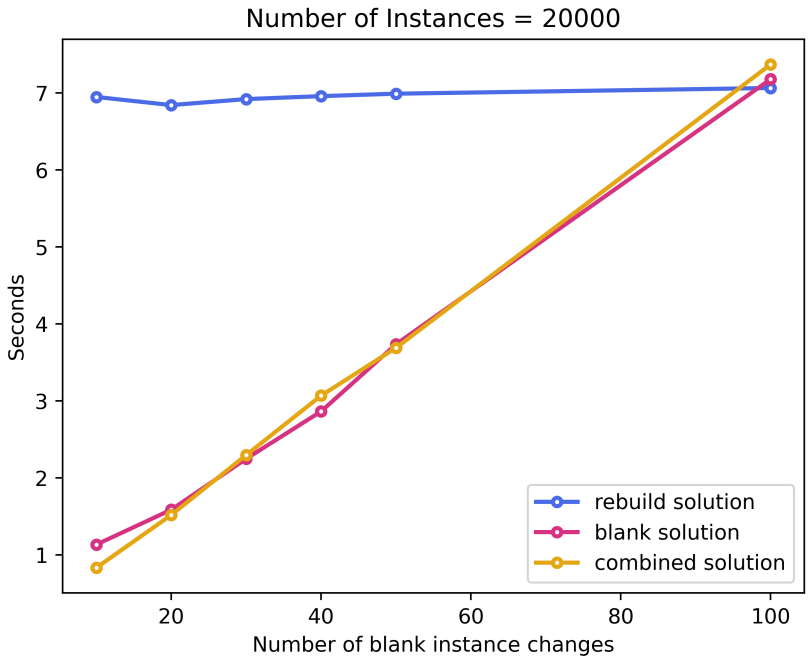

Figure 10.3: Combined solution, Blank node Solution, and Rebuild solution varying number of blank instance changes

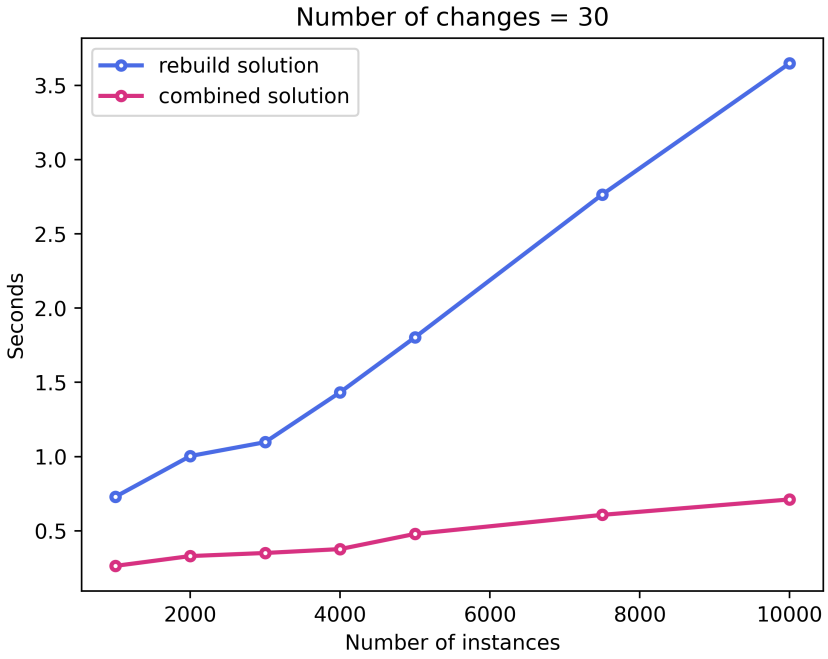

Figure 10.4: Combined solution vs. Rebuild solution combined example

10.3 Discussion

We will now briefly discuss the results of the experimental evaluation.

While the *Combined solution* is a more complicated algorithm, every changed triple will be handled only by the methods one of the previous solutions. Non-blank triples are handled by the *Duplicate solution*, while blank triples are handled by the *Blank node solution*, but never both. As discussed in the previous section, we can ignore duplicate blank triples, causing the two parts of the algorithms to be separate. The results in Figure 10.2 and Figure 10.3 show us, as expected, that the *Combined solution* performs similarly to the previous solutions in their respective intended use cases.

We assume that as the number of changes increases, the *Combined solution* will have a diminishing performance, compared to the *Rebuild solution*, especially with the deletion of blank instances. This is the trend of all previous solutions. However, it is reasonable to believe that most updates of OTTR data will still see an improvement in update speed with the *Combined solution*. Since it performs similarly to the other solutions, it severely outperforms rebuilding in the typical case from Section 5.1, but is heavily affected by deletion of blank instances.

It is important to note that without the assumption about top-level blank nodes, this solution would look different. The absence of top-level blank nodes allows us to ignore duplicates of blank triples. Without this assumption, we would have to combine the delete queries of blank nodes, with the counting algorithm.

As the *Combined solution* handles both blank instances and duplicate instances, it is more complete than the other solutions. Top-level blank nodes is the only kind of input data it does not handle. However, we believe that top-level blank nodes are uncommon and it should be easy to recognize they are present.

Part III

Discussion and conclusion

Chapter 11

Comparison

Comparing the different solutions to each other is a helpful tool in gauging their relative performance, especially compared to our baseline, the *Rebuild solution*. The various solutions operate under different assumptions, which means that not every solution can be applied to every test case. In this chapter, we look at how the different solutions compare.

The table in Figure 11.1 shows what types of operations the different solutions support. We see that all solutions support insertions of any kind of triple. The *Simple solution* has the strictest assumptions. The *Blank node solution* and *Duplicate solution* remove one assumption each. The *Combined solution* can handle the deletion of normal, blank, and duplicated triples. None of the solutions presented in this thesis can handle blank nodes being used as a top-level argument, as we in Section 5.1 reasoned that this is not common in a practical use case.

ATTRIBUTES	REBUILD SOLUTION	SIMPLE SOLUTION	DUPLICATE SOLUTION	BLANK NODE SOLUTION	COMBINED SOLUTION
INSERT ALL TYPES OF TRIPLES	✓	✓	✓	✓	✓
DELETE NORMAL TRIPLES	✓	✓	✓	✓	✓
DELETE BLANK TRIPLES	✓	✗	✗	✓	✓
DELETE DUPLICATE TRIPLES	✓	✗	✓	✗	✓
DELETE TOP LEVEL BLANK NODES	✓	✗	✗	✗	✗

Figure 11.1: Attributes of all solutions

When comparing the different solutions in a test with few changes and many instances, we see that all presented solutions significantly outperform the *Rebuild solution*. This is illustrated in Figure 11.2 where five instances are deleted, and five are inserted affecting only non-blank and non-duplicated triples. Note that the time in the figure is log scaled.

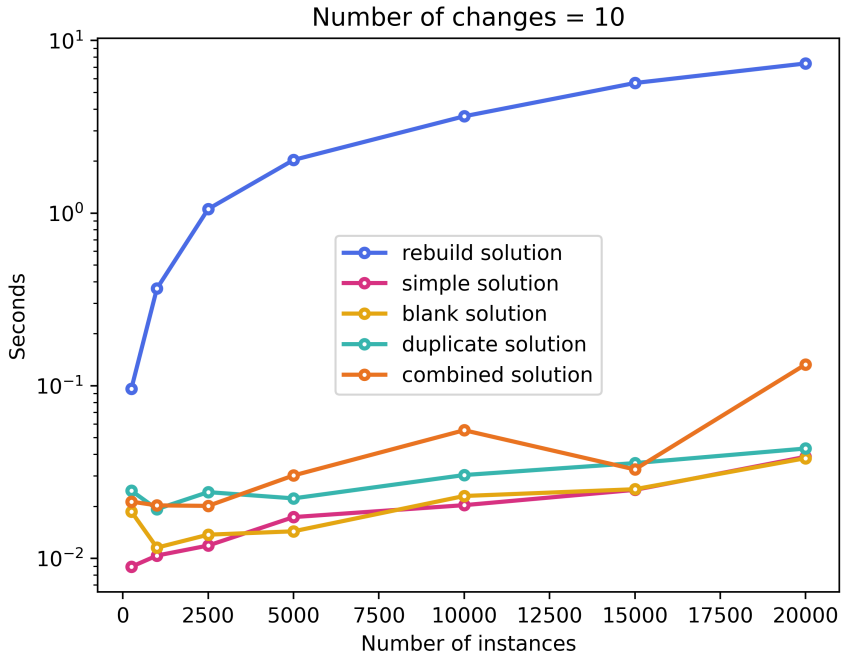

Figure 11.2: Typical case comparison. Log scale

Figure 11.3 and Figure 11.4 shows how the different solutions scale when the number of instances stays constant at 20000, and the number of deletions or insertions increases. Both graphs are log scaled, meaning the time spent by *Rebuild solution* is significantly larger than the other solutions when only non-duplicated and non-blank instances are affected. We also see that the *Duplicate solution* and the *Combined solution* are consistently slower than the *Blank node solution* and *Simple solution*.

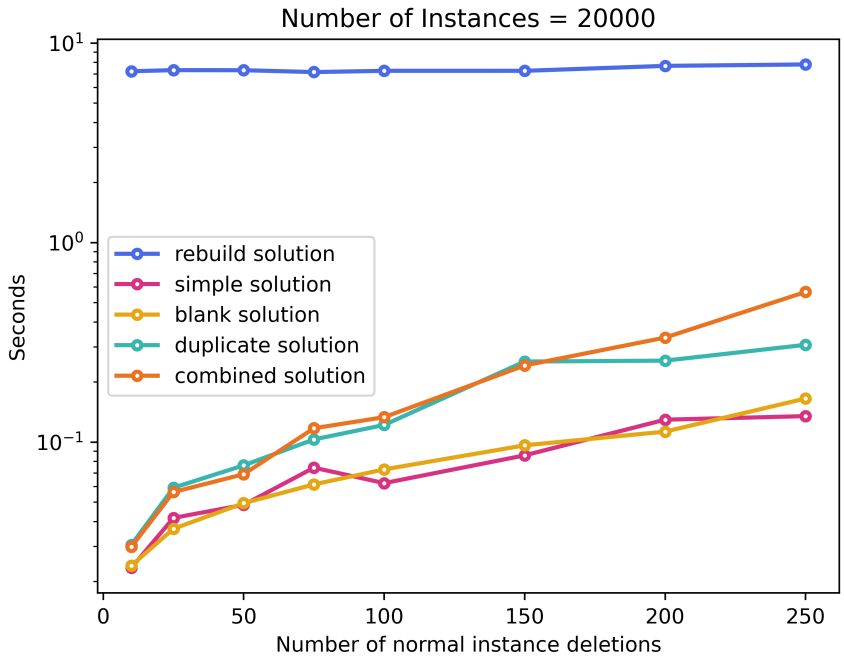

Figure 11.3: Only delete normal triples

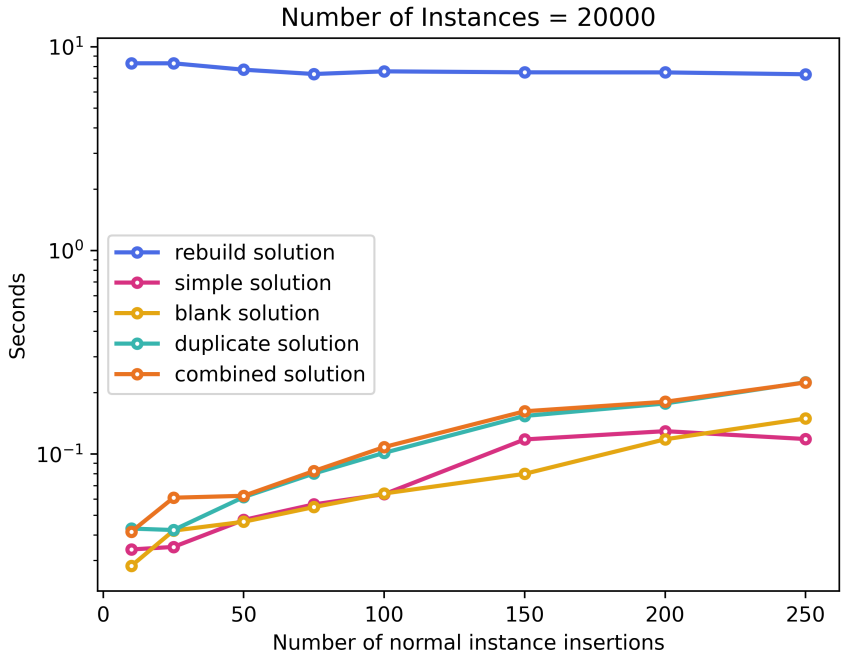

Figure 11.4: Only insert normal triples

Comparing how efficiently the *Blank node solution* handles blank instances with how efficiently the *Duplicate solution* handles duplicated instances is insufficient for saying anything about their relative performance. However, this comparison can tell us what types of changes are the most time-consuming. In Figure 11.5, we compared the different solutions deleting the type of instance they are created for. This means we look at the time spent by the *Blank node solution* deleting 5 to 100 blank instances. This is compared to the *Duplicate solution* deleting the same number of duplicate instances. Additionally, we compare this with the *Simple solution* deleting the same number of normal instances. We see clearly that deleting a blank instance is slower than deleting a duplicate instance or a normal instance. Figure 11.6 shows a similar graph, but in this case, the solutions are only inserting as opposed to only deleting. Here we see that the solutions scale similarly, with the duplicate solutions being somewhat slower than the other two. The type of instance that is inserted does not have a significant effect on the time spent.

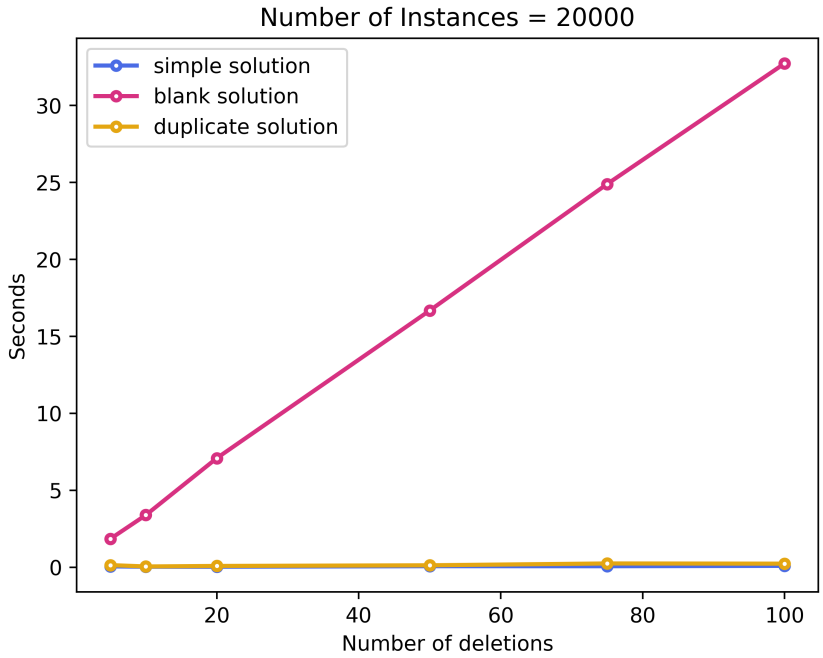

Figure 11.5: The three solutions deleting instances of their special case

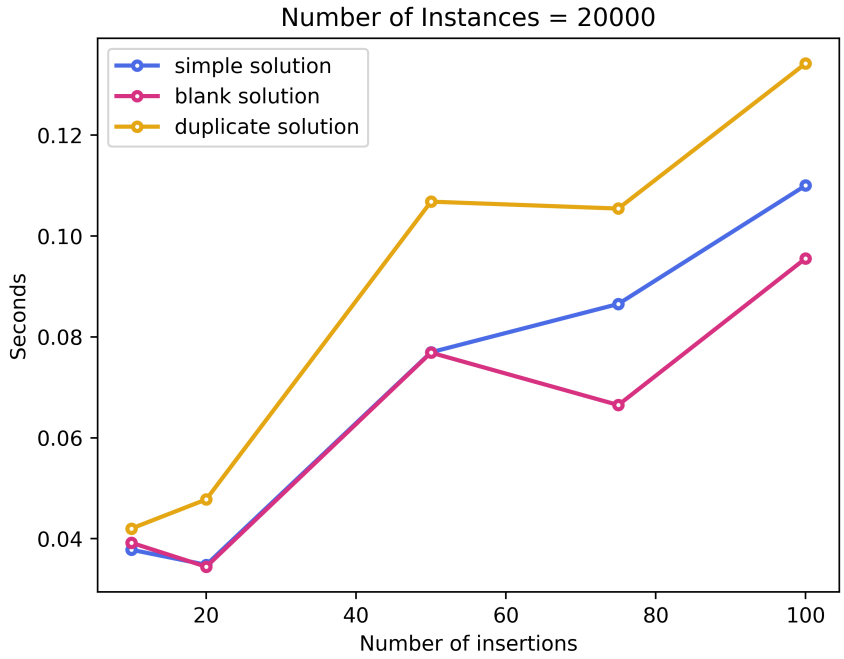

Figure 11.6: The three solutions inserting instances of their special case

Chapter 12

Discussion

We will now discuss the key findings of our results. Then, move on to how this research will affect OTTR, and discuss some considerations that must be accounted for in an implementation. Further, we will take a look at the limitations of our experiments. And lastly, we present some user recommendations.

12.1 Key findings

It is possible to create an efficient update algorithm for OTTR

Our algorithms will not perform better than rebuilding in all situations. However, all of our algorithms consistently outperform rebuilding in the typical cases discussed in Section 5.1. Given some assumptions, the *Simple solution* makes an efficient update. The *Duplicate* and *Blank node solution* show that it is possible to remove those assumptions and still have an efficient update algorithm. And the *Combined solution* is a fairly complete update algorithm, with the exception of handling top-level blank nodes.

Rebuilding is better with large changes

In contrast to the solutions presented in this thesis, the *Rebuild solution* is unaffected by the number of changes. It will always expand all OTTR instances and insert the resulting graph. This is very inefficient when the update is small, but this is the optimal solution if all instances are updated. When all instances are updated, our solutions would have to expand all instances in addition to other operations, which means the *Rebuild solution* would perform better. Thus, we know that there is a point where rebuilding the entire triplestore is more efficient than our solutions.

The point where this happens depends on the solution, the triplestore, the number of blank nodes and duplicates, and the changes made. Based on the results in Chapter 6, Chapter 8, Chapter 9, Chapter 10 and Chapter 11, we can conclude that the change has to be of significant size in comparison to the triplestore to reach this point. However, we believe such an update is unlikely when the triplestore is large, as the update would be very big.

An update algorithm without special cases is intuitive and efficient

During development, we found it easy to create an efficient update algorithm that does not take into account special cases like duplicates and blank nodes. The *Simple solution* is a very intuitive approach to the problem, and we see from the results that the performance is excellent. However, this solution has very strict assumptions, restricting its practical use.

Insertion is easier than deletion

We can see from Figure 11.1 that all solutions can insert any type of triple without desynchronizing the OTTR instances and the triplestore. Problems with blank nodes and duplicates only occur during deletion. If we only insert triples and avoid deletion, the *Simple solution* will work with both blank triples and duplicates. That means the *Simple solution* is our best choice if only inserting and non-destructive operations are allowed.

Counting duplicates affects both the insert and delete operation

In our *Duplicate solution*, the approach is to keep track of the number of duplicates. That way, we ensure we do not delete a triple with one or more duplicates. However, even as the desynchronization only happens in the delete operation, the insert operation must also be modified to check if new duplicates are added. Both the insertion and deletion operation must find specific resources in the triple store, meaning the *Duplicate solution* is slower at inserting than the other solutions. Still, it is much more efficient than rebuilding a big triplestore. The *Combined solution* uses the same method for inserting non-blank triples and shows a similar performance.

Matching blank nodes is a complex problem

While inserting blank nodes is easy, finding and deleting the correct ones is expensive. A blank node has no identifier outside a local scope, meaning we must find it by matching a pattern. That means using variables to check all combinations of possible values for the blank nodes. This is

expensive and time-consuming in large triplestores. Finding a triple with no variables should be easier, especially if indices are used, as there is only one possible value for each resource. Our data support this theory since it shows that deleting triples containing blank nodes is much more expensive than deleting normal triples. The delete operation in the *Blank node solution* scales not only with the size of the update, like the *Simple* and *Duplicate solution* mostly do, but also with the size of the triplestore and the number of blank nodes in the update. The *Blank node solution* must for each blank node in the delete query search for a pattern in the entire triplestore. However, deletion of normal triples without blank nodes is performed like in the *Simple solution*, not affected significantly by the size of the triplestore, as we can see in Figure 11.3. The problem of finding blank nodes is proved to be NP-hard [53].

Blank node solution is an extension of the Simple solution, Duplicate is different

Normal triples, which are non-duplicated triples without blank nodes, are handled very efficiently by the *Simple solution*. This is also the case for the *Blank node solution*, as normal triples are handled in almost the same way. Since the *Blank node solution* also handles blank nodes, it is an extended version of the *Simple solution*. Inserting and deleting triples with the *Duplicate solution* requires checking if the triple is a duplicate and acting accordingly. Both operations consist of checking if the triple is a duplicate and keeping track of the number of duplicates for each triple. We have to do this for all triples, duplicate or not, which results in a performance loss for all operations. This method is fundamentally different from the *Simple solution* and *Blank node solution*.

Blank triples do not have to be counted

In Chapter 10, we discovered that it was sufficient to use the methods from the *Blank nodes solution* when handling blank triples, even when duplicates are allowed. Our solutions always treat the whole expansion of a changed instance, rather than specific triples. Thus, duplicates within an instance can be ignored. We also have an assumption that ignores top-level blank nodes, which means a single blank node cannot occur in the expansion of different instances. If we combine these two facts, then we know that duplicates of blank triples do not need to be accounted for.

All solutions should rebuild when the update is too large

Our solutions perform well with small updates, but are outperformed by the *Rebuild solution* with very large updates. This is because rebuilding is

independent of the update, while our solutions are not. The ideal solution could be to switch to a rebuild approach if the update is large enough. However, this point is not at a consistent percentage, as it varies depending on the solution and the data. Thus, it would require specific testing for a specific scenario to find the optimal point to switch to rebuilding. Doing an analysis like this is interesting work that can be done in the future.

12.2 Implication for OTTR

This thesis gives insight into key challenges that must be taken into account and possible solutions. We also identify what factors affect the performance of such an algorithm. Implementing our algorithm would make OTTR a better fit for maintaining triplestores. Currently, OTTR's only option is to rebuild the entire triplestore, which in practice means using OTTR to update large triplestores is not a great choice as it is slow. This is not ideal since OTTR is intended for large amounts of data.

In this thesis, we have implemented the different solutions in Java using Lutra as a dependency to perform tests. Implementing the update algorithms directly in Lutra, in a similar manner, is achievable but comes with its own considerations. While we have explored how to handle changes to OTTR instances, it might also be desirable to handle changes to OTTR templates. Ideally, an implementation of this algorithm would address both of these cases. An alternative could be to rebuild when there is a change in the template files. This might be inefficient, but template changes are probably rarer while also being on average larger in size than instance changes. This is because template changes typically affect more triples than instance changes, and as updates scale with the size of the change, the template updates will probably not be as efficient as the instance updates.

Another consideration is the robustness of the algorithms. This thesis has provided a description and an example implementation of several update algorithms. Still, it has not addressed the issue of robustness, specifically with regard to handling errors during execution and input handling. This must be addressed for the update to be a part of OTTR. The most significant identified problem regarding our solution's robustness is that there is no check if the assumptions we have made are in place. It is essential to detect if an assumption, like the absence of blank nodes or the absence of syntax error, is maintained and to give an error if that is not the case. Another critical point to address is unexpected interactions with the triplestore. For example, if a connection to the database is not established, the algorithm must be robust enough to detect and handle this. All of the solutions perform multiple database request, and it is essential that they can handle if one or multiple of the requests fails.

Reasoning is another consideration. Reasoning over an RDF graph is a common use case, and this is still possible when using the update algorithm, but the derived triples should be stored in a separate graph. Every time we update the default graph with our algorithm, the derived triples must also be updated. This is done with truth maintenance [17], which is another type of update problem. We discuss the similarities between our update problem and truth maintenance in Chapter 13.

12.3 Limitations

When developing, testing, and evaluating an algorithm in a constructed environment, as in this thesis, there is no guarantee that the performance of an actual implementation will reflect our evaluations. The results presented may not be representative of an algorithm implemented in Lutra. Nevertheless, we believe the results would be similar since our implementation is implemented in Java using Lutra and Jena as libraries.

All of our tests are executed with the same setup of hardware and program versions. Our results do not show how the algorithms will perform with a different setup. For example, our tests used a locally hosted triplestore. This is likely not the case in a real-world scenario. Communicating with an online triplestore is much slower than with a local one, while query execution takes the same time. How this affects our solutions is not tested.

Our methodological assessment choices limit our research. This thesis uses a practical approach and evaluates how a solution compares to rebuilding in justified scenarios. However, the tests are not exhaustive. OTTR is intended as a general-use template language to create and maintain large ontologies and RDF graphs. The potential uses of the language are far too many to be feasibly tested. In this thesis, we have performed a variety of tests for each solution, where we have focused on the typical use case and the point where the *Rebuild solution* and the given solution perform approximately the same. Further testing of the solutions should be explored. An aspect of the algorithm that has not been adequately explored is its performance on very large data sets.

Synthetic test data impact the reliability of our results. All the tests in this thesis are executed on a modified set of Exoplanets [52]. The tests vary the number of instances, deletions, and insertions of normal triples, blank triples, and duplicate triples. This is done by programmatically changing the dataset. Using synthetic test cases like this to evaluate how an algorithm will perform during real-world use comes with uncertainty. Synthetic datasets do not always represent the real world, and our test will not capture the complexity, nuances, and patterns of actual usage scenarios.

12.4 User recommendations

Being aware of the strengths and weaknesses of the algorithm allows a user to make decisions that can increase performance. Therefore, studying how the algorithm performs in different scenarios can give valuable information on how to adjust the input data or the use of the algorithm to your specific use case.

Several considerations can be made to facilitate the efficient use of the OTTR update. As we have discussed, deleting blank nodes is the most complex operation, and thus reducing the need for deleting blank nodes will improve performance. This can be a hard requirement to follow, and a more manageable goal might be to reduce the number of blank nodes in general. We also know that allowing instances that create duplicate triples will require multiple HTTP requests on both the insert and delete operations, even when handling non-duplicate triples. If it is possible to ensure no duplicates exist, using only the *Simple solution* or *Blank node solution* will result in better performance. However, both the duplicate and the blank node problem only occurs when deleting, which means that if insertion is the only required operation, then the *Simple solution* will be sufficient.

The considerations will be different when using OTTR to create ontologies meant for reasoning. As we have discussed in Section 5.1, a dataset with a large TBox results in many duplicates. We know that the *Duplicate solution* only performs slightly worse when handling duplicated triples than normal triples. This means there is no need to limit the number of duplicates created to increase performance. However, you can expect the storage space to increase, as many counters are stored.

As a user, it is important to identify the requirements. Are you creating what is effectively a database with empty TBox, or are you creating a reasoning ontology? Does the algorithm need to handle duplicates and blank nodes, and is there a need for 100% correctness at all times? If one can tolerate some temporary inaccuracies and duplicates are rare, then one can use the *Simple solution* and occasionally rebuild to correct the inaccuracies. If 100% correctness is always required, this is not an option.

Chapter 13

Related Work

In this chapter, we will look at related work and discuss similarities and differences to our thesis. To our knowledge, the problem addressed in this thesis has not been addressed before.

13.1 A Truth maintenance system

A general truth maintenance system is presented in Jon Doyles *A Truth Maintenance System* [25]. The system makes assumptions and revises beliefs with new discoveries. This is done by recording and maintaining justifications for the beliefs. This system is similar to the algorithms presented in this thesis if we view the RDF graph as the set of beliefs and the justifications as a pair containing an RDF triple and the corresponding OTTR instance.

Our algorithm is more straightforward than the system presented by Doyle, as introducing new instances can only result in triples being added, not removed. In a truth maintenance system, this would be equal to adding new justifications only resulting in adding beliefs, not removing them. Adding new triples requires minimal logic, just expansion and insertion.

The deletion part of our process bears more resemblance to the truth management system. In *Inferencing and Truth Maintenance in RDF Schema* [17], Jeen Broekstra and Arjohn Kampman creates an algorithm, that is based on *A Truth Maintenence System*, for updating derived statements based on deletion of explicit statements. They aim to outperform a brute-force approach, where all derived statements are discarded and recomputed. Their problem is similar to ours, especially deleting duplicate triples, as their algorithm handles the removal of duplicate derived statements. Both their problem and our duplicate problem occur only when information is retracted.

In Broekstra and Kampman's algorithm, they store a set of justifications. A justification contains a derived statement and the justifying facts. They use the justifications to compute the consequences of removing an explicit statement. Removing a statement also removes the justifications this statement justifies. All statements derived from a deleted justification are not necessarily deleted, since there might still exist other justifications for those statements. However, if there is no justification for a derived statement, then it is deleted.

The justifications show why all derived statements is generated. In our case, a mapping like the justifications would be a mapping between RDF triples and OTTR instances. This would solve both the duplicate problem and the blank node problem. Unfortunately, this approach would imply storing a justification for every triple, which doubles the required space. As OTTR users are likely to use a large triplestore, doubling the required space might not be desirable.

13.2 Truth maintenance in datalog

Datalog is a database query language based on the logic programming paradigm [18]. Datalog implementations often *materializes* all facts entailed by a datalog program. This means, from a set of explicit facts, it pre-computes and stores all entailed facts. This is done in order to answer queries more efficiently. When the explicit facts are changed, all entailed facts must be rematerialized. *Materialization maintenance* algorithms have been developed to efficiently identify the required updates, to speed up rematerialization. Materialization maintenance shares many similarities with our problem if we view the explicit facts as the OTTR instances, and the materialized facts as the RDF triples. When a change to the explicit facts occurs, we want to materialize only the affected entailed facts.

The paper *Maintaining views Incrementally* by Gupta et al. [29] looks at the problem of changing materialized database views, in response to changes in the relations. They present a *counting* algorithm to compute updates to materialized views, by keeping track of the number of alternative derivations for each tuple. A change is seen as a set of changes, where tuples with a positive count represent insertions, and tuples with a negative count represent deletions. Applying a change to a materialized view means updating the count, and possibly inserting or deleting tuples. This approach is similar to the *Counting all occurrences* approach, discussed in Section 8.2, where all triples have a corresponding counter triple, and inserting or deleting triples entails changing the counter. The *Duplicate solution* implemented in this thesis only counts duplicates as opposed to all occurrences. This approach still resembles the *counting* algorithm presented by Gupta et al., as a triple without a count implies one occurrence of that triple.

Another approach to materialization management is the *Delete and Rederive (RDed) algorithm*. This algorithm is originally presented in the same paper *Maintaining views Incrementally* [29], but many variations and optimizations have been presented[37, 50]. The RDed algorithm makes changes to view relations given a change to the base relation. The algorithm does this in three steps. First, the algorithm computes an overestimate of the deleted derived tuples. Second, the overestimate is removed by looking for alternative derivations for the deleted derived triples. Finally, the new tuples that need to be added are computed using the partially updated materializations [29]. Applying a similar technique to remove the duplicate assumption could look like this: We first overestimate the triples to delete by deleting the whole expansion of the OTTR instances to delete. Secondly, we remove the overestimate by looking for any existing instances that create every triple to be deleted. Lastly, we add the expansion of the instances to be inserted. The main drawback of this approach, is the lack of a way to identify the corresponding OTTR instances to a given triple. It would take too much time to expand all instances for this check, and a mapping structure from instances to triples would require too much space, as we mentioned when discussing *A Truth Maintenance System* and *Inferencing and Truth Maintenance in RDF Schema*.

13.3 Other mapping languages

Another RDF mapping language similar to OTTR, is R2RML [43]. R2RML is a language for expressing mappings from relational databases to RDF. This is used to view relational data as RDF, for example with a virtual SPARQL endpoint. The mappings are written in RDF turtle format and specify how the relational data should be translated into RDF. Users can also customize the mappings. R2RML has several similarities to OTTR, as OTTR can also be used to translate relational data to OTTR instances with bOTTR mappings, and then further translated to RDF via expansion. However, R2RML has no reason for having an update algorithm, as it only creates virtual triplestores. This means that there is nothing to update based on changes in the relational database.

RML [23] is another mapping language based on and extending R2RML, that can be used for materializing triplestores. RML supports a broader range of inputs compared to R2RML, which only uses a relational database as input. However, to our knowledge, RML does not handle updates of source data [15, 23].

Chapter 14

Conclusion

In this thesis, we have created multiple algorithms for updating a triplestore based on changes to an OTTR instance file. The algorithms outperform OTTR's current solution in typical use cases, which is a small change to a large triplestore. Our update algorithms all use a difference algorithm to identify changed OTTR instances and then create a corresponding SPARQL query to update the triplestore.

We have shown that it is possible to create an efficient update algorithm for OTTR. The different algorithms are created for different use cases with varying assumptions. The *Simple solution* performs the best, but under the strictest assumptions, as no OTTR instances that create triples with blank nodes can be deleted (*local blank node assumption*), and no OTTR instances that create duplicate triples can be deleted (*duplicate assumption*). The *Blank node solution* removes the *local blank node assumption*. It performs well with non-blank triples and insertion of blank triples, but deleting triples with blank nodes scales poorly. The *Duplicate solution* removes the duplicate assumption by keeping track of the number of duplicates. Compared to the *Simple solution*, we see that it scales similarly, but is generally slower. Lastly, the *Combined solution* combines the three algorithms mentioned above to remove both the *duplicate assumption* and the *local blank node assumption*. This algorithm performs similarly to previous solutions at their respective input type, as it uses the *Blank node solution* on triples containing blank nodes, and the *Duplicate solution* on all other triples. In a typical use case with a small change to a big triplestore, all implemented solutions outperform OTTR's current solution, which is to rebuild the entire triplestore.

We have seen that the input data affect the performance of the algorithm. The smaller the update is relative to the triplestore, the better our solutions perform compared to rebuilding. The inclusion of blank nodes affects performance, as deleting triples with blank nodes requires costly graph matching in the triple store. Inserting non-duplicated triples is effective

in comparison. Allowing OTTR instances to create duplicates affects all operations, also when handling non-duplicates, as all operations need to check if a triple is a duplicate.

We believe our algorithms would be a good addition to OTTR, as the inclusion of our algorithms would make OTTR a language for maintaining triples stores in addition to creating RDF graphs. There is no update in OTTR's current state, and the only option is to rebuild the entire RDF graph in the triplestore. In many cases, rebuilding would not be feasible, as it can be very time-consuming. Our update algorithms allow OTTR to make small changes to a large triplestore efficiently. This is a required functionality since OTTR intends to improve the efficiency and quality of maintaining knowledge bases. However, several considerations must be taken into account before adding this as part of OTTR. It is important that the update is sound and does not produce incorrect updates. A more theoretical assessment of this work to ensure soundness should be in place. Additionally, ensuring that the implementation is robust and capable of handling errors is important.

14.1 Future work

We propose the following future work related to OTTR and updates:

- **Implementation in Lutra**
  A natural next step from this thesis would be implementing the update algorithm as a part of Lutra and the OTTR language, not only as a standalone test program.
- **Template update problem**
  While this thesis looks at handling updates of changed OTTR instances, another variation worth exploring is the case where the OTTR templates have been changed. A solution to this can be built upon the algorithm presented in this thesis, as it supports both inserting and deleting instances. A simple approach to handling a change of a template can be: expand the instances over the old template and delete the result, then expand the potentially modified instances over the new template and insert these.
- **Explicitly handle modified instances**
  A likely scenario is when an update consists of modifying one or more instances, not only deleting or adding. This thesis handles modified instances by deleting the old version and inserting the new version. Handling these cases explicitly is interesting to explore, as unnecessary deletions and insertions can be avoided, especially if a single OTTR instance expands to a large number of triples.

- **Theoretical approach**
  While this thesis took a practical approach to solving the problem and evaluating the results, it is also interesting to analyze the theoretical aspects. Proof of correctness for each solution presented in this thesis can be an interesting continuation of this work. In addition, other theoretical aspects, such as what an optimal update is, can also be explored.

- **More extensive testing**
  There is a need for more extensive testing of the solutions presented in this thesis. This testing can vary more variables, for example, the testing system or other test cases.

- **Updates specialized for large TBox ontologies**
  We have seen that the update algorithm presented in this thesis performs the worst when there are many blank nodes and duplicates. This typically occurs when the ontology is used for reasoning, and the TBox is of significant size, as discussed in Section 5.1. Investigating alternative ways to create the update algorithm in order to perform better in these scenarios is of interest.

- **bOTTR updates**
  One way OTTR is used to create RDF graphs is by mapping a SQL database to OTTR instances by using bOTTR [48]. Exploring a way of creating a correct update to the triplestore by knowing the change to the SQL database and the bOTTR mapping can be of interest.

- **Unknown Update Problem**
  Another problem variation is the case where the update of instances and templates is unknown. This problem occurs when we have new versions of OTTR templates and instances but not the old versions. This makes it harder to identify changes, which makes it harder to create an update query, but it removes the need to keep track of old and new versions of the instance file.

Bibliography

[1] *About W3C*. [Online; accessed 11. May 2023]. Feb. 2023. URL: https://www.w3.org/Consortium.

[2] *Abox - Wikipedia*. [Online; accessed 24. Apr. 2023]. Mar. 2022. URL: https://en.wikipedia.org/w/index.php?title=Abox&oldid=1079348696.

[3] *Aibel*. [Online; accessed 29. Apr. 2023]. Apr. 2023. URL: https://aibel.com/company.

[4] Carlos Buil Aranda et al. *SPARQL 1.1 Overview*. [Online; accessed 26. Apr. 2022]. Oct. 2018. URL: https://www.w3.org/TR/2013/REC-sparql11-overview-20130321.

[5] David Peterson et al. *w3c xml schema definition language (xsd) 1.1 part 2: datatypes*. 2012. URL: https://www.w3.org/TR/xmlschema11-2/.

[6] Eric Prud'hommeaux et al. *RDF 1.1 Turtle*. [Online; accessed 5. Apr. 2022]. Jan. 2018. URL: https://www.w3.org/TR/turtle.

[7] Olaf Hartig et al. *RDF-star and SPARQL-star*. [Online; accessed 7. Feb. 2023]. Dec. 2022. URL: https://w3c.github.io/rdf-star/cg-spec/editors_draft.html#sparql-star.

[8] Olaf Hartig et al. *RDF-star and SPARQL-star — GraphDB 10.0.0 documentation*. [Online; accessed 20. Apr. 2023]. Mar. 2023. URL: https://graphdb.ontotext.com/documentation/10.0/devhub/rdf-sparql-star.html.

[9] Richard Cyganiak et al. *RDF 1.1 Concepts and Abstract Syntax*. 2014. URL: https://www.w3.org/TR/rdf11-concepts/.

[10] Richard Cyganiak et al. *RDF 1.1 Concepts and Abstract Syntax*. [Online; accessed 4. Apr. 2022]. 2018. URL: https://www.w3.org/TR/rdf11-concepts/#section-IRIs.

[11] *Apache Jena - Support of RDF-star*. [Online; accessed 28. Feb. 2023]. Feb. 2023. URL: https://jena.apache.org/documentation/rdf-star.

[12] *Apache Jena - Support of RDF-star*. [Online; accessed 20. Apr. 2023]. Apr. 2023. URL: https://jena.apache.org/documentation/rdf-star.

[13] *Apache Jena - The core RDF API*. [Online; accessed 5. May 2023]. Apr. 2023. URL: https://jena.apache.org/documentation/rdf/index.html.

[14] Catriel Beeri, Philip A Bernstein and Nathan Goodman. 'A sophisticate's introduction to database normalization theory'. In: *Readings in artificial intelligence and databases*. Elsevier, 1989, pp. 468–479.

[15] Thomas Delva Ben De Meester Pieter Heyvaert. *RDF Mapping Language (RML)*. [Online; accessed 10. May 2023]. Nov. 2022. URL: https://rml.io/specs/rml.

[16] *BlankNodes - W3C Wiki*. [Online; accessed 26. Apr. 2023]. Apr. 2023. URL: https://www.w3.org/wiki/BlankNodes.

[17] Jeen Broekstra and Arjohn Kampman. 'Inferencing and Truth Maintenance in RDF Schema.' In: *PSSS* 89 (2003), p. 10.

[18] Stefano Ceri, Georg Gottlob, Letizia Tanca et al. 'What you always wanted to know about Datalog(and never dared to ask)'. In: *IEEE transactions on knowledge and data engineering* 1.1 (1989), pp. 146–166.

[19] X/Open Company. 'Technical Standard Data Management: Structured Query Language (SQL) Version 2'. In: (Jan. 2003). [Online; accessed 5. Apr. 2022]. URL: https://pubs.opengroup.org/onlinepubs/9695959099/toc.pdf.

[20] *Comparing and Merging Files*. [Online; accessed 25. Apr. 2022]. Apr. 2022. URL: https://www.gnu.org/software/diffutils/manual/diffutils.html.

[21] R.V. Guha Dan Brickley. *RDF Schema 1.1*. [Online; accessed 7. Apr. 2022]. Oct. 2017. URL: https://www.w3.org/TR/rdf-schema.

[22] *Data - W3C*. [Online; accessed 23. Apr. 2023]. June 2019. URL: https://www.w3.org/standards/semanticweb/data.

[23] Anastasia Dimou et al. 'RML: A generic language for integrated RDF mappings of heterogeneous data.' In: *Ldow* 1184 (2014).

[24] dotNetRDF Project. *dotNetRDF*. [Online; accessed 30. Mar. 2023]. Mar. 2023. URL: https://dotnetrdf.org.

[25] Jon Doyle. 'A truth maintenance system'. In: *Artificial Intelligence* (Nov. 1979). URL: https://doi.org/10.1016/0004-3702(79)90008-0.

[26] Andy Seaborne Eric Prud'hommeaux. *SPARQL Query Language for RDF*. [Online; accessed 1. May 2023]. Oct. 2018. URL: https://www.w3.org/TR/rdf-sparql-query/#construct.

[27] *Git - git-diff Documentation*. [Online; accessed 25. Apr. 2022]. Apr. 2022. URL: https://git-scm.com/docs/git-diff.

[28] Google. *diff-match-patch*. [Online; accessed 25. Apr. 2022]. Apr. 2022. URL: https://github.com/google/diff-match-patch.

[29] Ashish Gupta, Inderpal Singh Mumick and Venkatramanan Siva Subrahmanian. 'Maintaining views incrementally'. In: *ACM SIGMOD Record* 22.2 (1993), pp. 157–166.

[30] Olaf Hartig. *Position Statement: The RDF∗ and SPARQL∗ Approach to Annotate Statements in RDF and to Reconcile RDF and Property Graphs*. [Online; accessed 29. Apr. 2023]. Apr. 2023. URL: https://blog.liu.se/olafhartig/2019/01/10/position-statement-rdf-star-and-sparql-star.

[31] Olaf Hartig and Bryan Thompson. 'Foundations of an Alternative Approach to Reification in RDF'. In: *arXiv* (June 2014). DOI: 10.48550/arXiv.1406.3399. eprint: 1406.3399.

[32] Martin G. Skjæveland Leif Harald Karlsen. *mOTTR: Concepts and Abstract Model for Reasonable Ontology Templates*. [Online; accessed 25. Apr. 2022]. Mar. 2019. URL: https://spec.ottr.xyz/mOTTR/0.1.

[33] Preben Zahl Magnus Wiik Eckhoff. *OTTR-updates - Github repo*. May 2023. URL: https://github.com/Masteroppgave-ottr/OTTR-updates.

[34] Daniel Lupp Martin G. Skjæveland Leif Harald Karlsen. *pOTTR: Reasonable Ontology Templates Fundamentals*. [Online; accessed 9. May 2022]. Apr. 2020. URL: https://primer.ottr.xyz/01-basics.html.

[35] *Model (Apache Jena API)*. [Online; accessed 29. Mar. 2023]. Jan. 2023. URL: https://jena.apache.org/documentation/javadoc/jena/org.apache.jena.core/org/apache/jena/rdf/model/Model.html.

[36] Jeffrey Mogul et al. *Delta encoding in HTTP*. Tech. rep. 2002.

[37] Boris Motik et al. 'Maintenance of datalog materialisations revisited'. In: *Artificial Intelligence* 269 (2019), pp. 76–136.

[38] Eugene W Myers. 'AnO (ND) difference algorithm and its variations'. In: *Algorithmica* 1.1 (1986), pp. 251–266.

[39] *owl 2 web ontology language structural specification and functional-style syntax (second edition)*. 2012. URL: https://www.w3.org/TR/owl2-syntax/.

[40] Axel Polleres Paula Gearon Alexandre Passant. *SPARQL 1.1 Update*. [Online; accessed 12. Apr. 2022]. Mar. 2013. URL: https://www.w3.org/TR/2013/REC-sparql11-update-20130321.

[41] *RDF-star Working Group*. [Online; accessed 5. May 2023]. May 2023. URL: https://www.w3.org/groups/wg/rdf-star.

[42] *RdfReification - W3C Wiki*. [Online; accessed 5. May 2023]. May 2023. URL: https://www.w3.org/wiki/RdfReification.

[43] Mariano Rodriguez-Muro and Martin Rezk. 'Efficient SPARQL-to-SQL with R2RML mappings'. In: *Journal of Web Semantics* 33 (2015), pp. 141–169.

[44] Jack Rusher. *Rhetorical Device*. [Online; accessed 1. May 2023]. Nov. 2003. URL: https://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/rusher.html.

[45] *Semantic Web - W3C*. [Online; accessed 23. Apr. 2023]. Mar. 2023. URL: https://www.w3.org/standards/semanticweb.

[46] Martin G Skjæveland. *Reasonable Ontology Templates (OTTR)*. 2021. URL: https://ottr.xyz/#Specifications.

[47] Martin G Skjæveland et al. 'Semantic Material Master Data Management at Aibel.' In: *ISWC (P&D/Industry/BlueSky)*. 2018.

[48] Martin G. Skjæveland. *bOTTR: Batch instantiation of OTTR templates*. [Online; accessed 17. Apr. 2023]. Feb. 2023. URL: https://dev.spec.ottr.xyz/bOTTR.

[49] Martin G. Skjæveland. *Reasonable Ontology Templates (OTTR)*. [Online; accessed 4. May 2023]. Mar. 2023. URL: https://ottr.xyz/#Lutra.

[50] Martin Staudt and Matthias Jarke. *Incremental maintenance of externally materialized views*. Citeseer, 1995.

[51] *The Eclipse RDF4J Framework · Eclipse RDF4J™ | The Eclipse Foundation*. [Online; accessed 30. Mar. 2023]. Mar. 2023. URL: https://rdf4j.org/about.

[52] *The Extrasolar Planets Encyclopaedia*. [Online; accessed 25. Apr. 2023]. Apr. 2023. URL: http://exoplanet.eu.

[53] Yannis Tzitzikas, Christina Lantzaki and Dimitris Zeginis. 'Blank Node Matching and RDF/S Comparison Functions.' In: *ISWC (1)*. 2012, pp. 591–607.

[54] *UpdateBuilder (Apache Jena Query Builder)*. [Online; accessed 29. Mar. 2023]. Jan. 2023. URL: https://jena.apache.org/documentation/javadoc/extras/querybuilder/org/apache/jena/arq/querybuilder/UpdateBuilder.html.

[55] *What is RDF-Star?* [Online; accessed 7. Feb. 2023]. Feb. 2021. URL: https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf-star.

[56] Antoine Zimmermann. *RDF 1.1: On Semantics of RDF Datasets*. [Online; accessed 16. Mar. 2023]. Oct. 2017. URL: https://www.w3.org/TR/rdf11-datasets/#options.