FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Updates in the Context of Ontology-Based Data Management

PhD THESIS

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

within the

Vienna PhD School of Informatics

by

MSc. Aljbin Ahmeti
Registration Number 1128387

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Prof. Axel Polleres

External reviewers:
Domenico Lembo. Sapienza University of Rome, Italy.
Werner Nutt. Free University of Bozen-Bolzano, Italy.

Vienna, 24th January, 2020

Aljbin Ahmeti	Axel Polleres

Declaration of Authorship

MSc. Aljbin Ahmeti
Address

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 24th January, 2020

Aljbin Ahmeti

To my daughter Arya, who is a Godsend.

Acknowledgements

I would like to express my sincere gratitude to my advisor Prof. Axel Polleres for his guidance, feedback and immense knowledge that helped me conduct the research, which resulted with this PhD thesis. I also thank him for giving me the opportunity to work as a researcher at the outstanding campus at WU Wien.

Besides my advisor, I would like to thank the thesis committee: Prof. Domenico Lembo and Prof. Werner Nutt, for their constructive comments as well as encouragement, which helped in polishing, and ultimately creating a much better presentation of the thesis from many different perspectives.

A special thanks goes to Dr. Vadim Savenkov, who was always supportive and available to answer my questions any time I was in need. Apart Vadim, I also thank my other co-authors: Prof. Diego Calvanese for his immense knowledge and his valuable input, especially during the early phase; Dr. Javier D. Fernández, with whom I had fruitful discussions at WU Wien; Dr. Simon Razniewski, who helped me to settle during my short spell at Free University of Bolzano. I was very lucky to meet all of you in person, to write papers, and to learn from each one of you. A very big thank you!

Next, I would take the opportunity to thank Prof. Nysret Musliu, with whom I spent a big chunk of my time while at TU Wien, and for his guidance that helped me tremendously, especially in the period when I started my PhD studies. Also, I would like to thank Prof. Reinhard Pichler for allowing me to join Database and Artificial Intelligence (DBAI) group as a PhD student, where I had the chance to learn a lot from the DBAI seminars. There, I would also thank an unsung hero - Toni Pisjak, who was very friendly and open to solve any technical questions that I had regarding the facilities.

Next, I thank all my colleagues at PhD School of Informatics for the great time spent together, for the ups and downs, and for everything we shared during our unforgettable time as PhD students. For this to happen, I have to thank both Clarissa Schmid and Maria del Carmen Moreno who did an outstanding job regarding the organisational aspects. A big shout out to the directors of PhD School for the hard work, and for the continuous strive to raise the bar of the PhD School in order to make it an ultimate model, namely to Prof. Hannes Werthner, Prof. Hans Tompits, and currently to Prof. Andreas Steininger.

Next, I would like to express my gratitude to Prof. Maurizio Lenzerini, who introduced me to the area of Data Integration/Semantic Web during my previous studies at Sapienza University of Rome. Thanks to his unforgettable lectures and our co-operation during my master thesis, that instilled passion in me for the subjects, I chose my PhD thesis in this area, and even now working as a professional at Semantic Web Company (SWC).

Next, I would like to thank all my colleagues at SWC, especially Helmut Nagy and Martin Kaltenböck, for their continuous support and for simultaneously making me feel bad when asking about my thesis ;-). I also thank Hakim Tafer for doing the first draft translation of the thesis' abstract into German language.

Last but not least, I would like to say a big thank you to my family: my parents and siblings for their endless motivational, financial and spiritual support, and to my wife Saranda for her patience, unconditional love, especially for taking care of our daughter Arya - while I was busy writing this thesis.

The research presented herein was supported by the Vienna Science and Technology Fund under the project SEE: ICT12-15 and the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the project Open Data for Local Communities (CommuniData, grant no. 855407).

Kurzfassung

Die Einführung von standardisierten *Vokabularen* und *Ontologien* sowie die Standardisierung der Abfragesprache *SPARQL* hat in den letzten Jahren zu einem starken Wachstum an im Web verfügbarer *strukturierter Daten* geführt. Ontologien als konzeptionelle, formale Repräsentation bestimmter Wissensdomäne, werden zur Beschreibung der Daten verwendet und bieten damit eine Grundlage für die Anwendung von Methoden des logischen *Schließens* über diese Daten. Sowohl Daten als auch Ontologien werden in diesem Semantischen Web als sogenannte RDF *Tripel* dargestellt und bilden so einen semantischen Graphen von Fakten und Axiomen. Sogenannte SPARQL Entailment Regimes spezifizieren das Zusammenspiel zwischen SPARQL-Abfragen und Ontologien formal, sodass als Antwort auf Abfragen auch implizite Tripel zurückgegeben werden.

Da strukturierte Daten im Laufe der Zeit, auch dank der Personen und Agenten, die sie unter Einhaltung der Linked-Open-Data-Prinzipien veröffentlichen, gewachsen sind, wurden Herausforderungen bezüglich der Datenverwaltung und -speicherung deutlich. Als Konsequenz daraus wurden verschiedene Datenmanagement Systeme für RDF entwickelt, so genannte *Triple Stores*, die in der Lage sind, Milliarden von Triples zu speichern und zu verwalten, sowie es erlauben, komplexe Abfragen unter Berücksichtigung von logischen Entailments zu beantworten. Solche Triple-Stores sind in der Lage, große Wissensbasen wie z.B. DBpedia, das sich im Zentrum der Linked Open Data Cloud befindet, zu speichern. Die strukturierten Daten in DBpedia werden über eine Reihe von *Mappings*, sogenannte Extraktoren, aus Wikipedia-Infoboxen extrahiert. Die Datenbasis von DBpedia kann von diversen Anwendungen konsumiert mit Hilfe von SPARQL-Abfragen über sogenannte SPARQL-Endpunkte konsumiert werden, die wiederum auf Triple-Stores aufgebaut sind. Dennoch geht die Vision des Semantischen Web einen Schritt weiter, das im Sinne nicht nur das “Lesen” sondern auch das “Schreiben” von strukturierten Daten zu ermöglichen; aus diesem Grund wurde in SPARQL auch eine Update Sprache definiert. Auffallend ist jedoch, dass die SPARQL Entailment Regimes Spezifikation des W3C das Zusammenspiel zwischen den SPARQL Updates und logischem Schließen über Ontologien nicht definieren.

Diese Dissertation zielt darauf ab, *verschiedene semantiken* für SPARQL Updates zu untersuchen, die über Daten im Zusammenspiel mit Ontologien gestellt werden, indem Ontologiesprachen bzw. Mappings unterschiedlicher Ausdrucksstärke berücksichtigt werden. Anders ausgedrückt, geht es darum, SPARQL Updates im Kontext von *Ontology-*

basiertem Datenmanagement (kurz: OBDM) zu untersuchen. Zunächst schlagen wir verschiedene Update Semantiken für Triple Stores vor, die Daten zusammen mit Ontologien speichern, die in der RDFS-Ontologiesprache ausgedrückt sind. Wir unterscheiden darin zwischen der Updates die nur Daten betreffen und solchen, die auch die Axiome der Ontologien betreffen, indem wir diese getrennt behandeln. Als nächsten Schritt untersuchen wir wir Updates in einer Ontologiesprache, die auch Axiome zu disjunkten Klassen und somit Inkonsistenzen zulässt, wobei wir in dieser Konstellation verschiedene, von der Belief Revision inspirierte Update Semantiken vorschlagen, die in der Lage sind, mit Inkonsistenzen in den Daten umzugehen. Zuletzt untersuchen wir verschiedene Update-Semantiken im vollen Rahmen von OBDM, d.h. dort, wo auch Mappings zu darunterliegenden, relationalen Datenquellen berücksichtigt werden und die Ontologiesprache noch ausdrucksstärker ist – wobei wir speziell DBpedia als Anwendungsfall behandeln.

Abstract

Nowadays, the amount of *structured data* on the Web has increased dramatically with the adoption of standardised *vocabularies* and *ontologies* and the standardisation of *SPARQL* query language. Ontologies as a conceptual, formal representation of a domain, are used to describe the data and, as a key functionality to provide *reasoning* capabilities. Both data and ontologies in the realm of the Semantic Web are represented as *triples* in RDF building a "semantic graph" of facts and axioms. SPARQL Entailment Regimes clearly specify the interplay between SPARQL queries and ontologies, so that as answer to queries implicit triples are returned as well.

As structured data have grown over time—also thanks to people and agents publishing them adhering to the Linked Open Data principles—questions of data management and storage became apparent. As a consequence, we have witnessed a development of bespoke RDF data management systems called *triple stores* capable of storing, managing and reasoning over billions of triples. Such triple stores are capable of storing big knowledge bases, for instance, DBpedia, being in the centre of the Linked Open Data cloud. The structured data in the case of DBpedia are extracted from Wikipedia infoboxes via a set of *mappings*, so-called extractors. DBpedia is a dataset that many applications consume by using SPARQL queries via so-called SPARQL endpoints, which in turn are established on top of triple stores. Nevertheless, the vision of the Semantic Web is to enable "Read/Write" Web of structured data; for this reason, the SPARQL Update language was introduced. Remarkably though, the SPARQL Entailment Regimes specification by the W3C does not define the interplay between the SPARQL updates and ontologies.

This dissertation aims to study different *update semantics* posed over data and ontologies, by taking into account different ontology language and mapping expressiveness respectively. In other words, the aim is to study SPARQL updates in the context of *Ontology-Based Data Management* (abbr. OBDM). First, we propose different update semantics for triple stores that store ontologies together with data, expressed in the RDFS ontology language. We distinguish therein between update semantics for data versus ontologies, by treating them separately. Then, as a next step, we increase the expressivity of the ontology language by adding class disjointness axioms, where in this setting we propose different update semantics inspired by belief revision that are able to handle inconsistencies in the data. In the end, we investigate different update semantics

in the full setting of OBDM, i.e., where also mappings to underlying relational data sources are considered and the ontology language is even more expressive – adopting DBpedia as a use case.

Contents

CHAPTER 1

Introduction

In the last years, the amount of structured data on the Web has increased dramatically. This growth originated partially from the standardisation efforts around the Semantic Web for representing structured data using the Resource Description Format (RDF)[Hay04] as underlying graph data format, and on the other hand due to Linked Data principles being put in practice[1][2] for publishing, connecting and discovering such data. Nowadays, both humans and agents are able to consume and publish structured data, thus slowly yet steadily approaching the Semantic Web vision [BLHL01].

In its core, the Semantic Web is about enabling "Read/Write"[3], i.e., reading and writing structured data for agents that could both be humans and machines. Naturally, having already the structured data in place, the necessity for an appropriate query and update language emerged and we have recently also witnessed the standardisation of SPARQL Query [HS13] and SPARQL Update languages [GPP13], respectively. The importance of SPARQL for querying and manipulating graph data is closely tied with the success of the Semantic Web, being an important factor on the emergence of RDF as well. There has been a well-established work in SPARQL querying and its interplay with the data schemata expressed in ontology languages such as RDFS [BGe04] and OWL [MGH+12] in the SPARQL 1.1 Entailment Regimes specification [GOH+13]. Nevertheless, still a major challenge remains on how to properly implement the "write" part, namely on how the interplay between updates and the data schemata takes part in the context of Semantic Web technologies, especially when legacy sources are mapped to RDF on the Web of data.

[1] https://www.w3.org/TR/ld-bp/
[2] https://www.w3.org/DesignIssues/LinkedData.html
[3] https://www.w3.org/community/rww/

1.1 Background

Today many of the datasets available in RDF—as witnessed by the Linked Open Data cloud[4]—are extracted by custom wrappers in a typical Extract Transform Load (ETL) process. These data are stored separately in dedicated databases and used mainly in a read-only fashion, i.e., data to be linked for consumption only. DBpedia [LIJ+14] being the focal point of this cloud to which other datasets are tightly interlinked, is often referred to as a core Semantic Web success story. The DBpedia dataset is generated using a custom set of wrappers having as input the key-value pairs from Wikipedia infoboxes, the largest online encyclopaedia. The English version of the DBpedia dataset as of year 2015 contains information about 4.8 million entities in the form of 176 million statements (triples) constituting the DBpedia graph[5].

Complementing the standardisation efforts for Semantic Web languages for representing structured data as mentioned previously, in different communities and various domains a number of *ontologies*[6] have been published on the Web of data, that is, axiomatic schema descriptions of the terms used in RDF and Linked Data. Ontologies formally describe the world (or a specific domain as part of the world) in terms of concepts and relationships between them. Therefore, using a standard, well-adopted ontology, one is able to describe the data, as well as to interlink concepts and terms appearing in the Linked Open Data cloud. On top of that, what is a distinguishing characteristic of ontologies in the context of Semantic Web standards compared to conventional schemata, is the ability to do *reasoning*, i.e., deriving implicit knowledge by using the pre-defined set of axioms encoded in the ontology. For instance, DBpedia has its own community-driven ontology enabling to derive new implicit data from the extracted data, which are not explicitly stated in Wikipedia itself.

It is of a common practice that data and ontology altogether are stored in a *knowledge base*. In the terminology of Description Logic (DL) [BCM+03] we refer to the instances of the data as *ABox*, and to the ontology as *TBox* respectively. The most common ABox and TBox services[7] are queries and updates, which are posed over the knowledge base and results are returned to the user in the case of queries. A number of systems, in no particular ordering: OWLIM [BKO+11] (now GraphDB), Sesame [BKVH02] (now RDF4J), Jena TDB[8], RDFox [NPM+15], Virtuoso[9], Stardog[10], Marklogic[11], Allegro-

[4] http://lod-cloud.net

[5] http://wiki.dbpedia.org/services-resources/datasets/dataset-2015-10/dataset-2015-10-statistics

[6] As yet another definition, one would say that an ontology is a collection of axioms describing the terms in a schema.

[7] Other important DL reasoning services are TBox classification, ABox and TBox satisfiability checking and so forth. For more details refer to [BCM+03].

[8] https://jena.apache.org/documentation/tdb/index.html

[9] https://virtuoso.openlinksw.com

[10] http://www.stardog.com

[11] http://www.marklogic.com

Graph[12], Blazegraph[13] (now Neptune) appeared recently that are known as *triple stores*. These systems are designed to store such knowledge bases and have a scalable architecture similar to database management systems (DBMS), but specialised for RDF triples. Typically, they serve as containers for both ABoxes and TBoxes, which both can be represented in terms of RDF triples. Such systems are able to store and manage billions of triples, offering standard services such as querying and updating as well as other features such as *parsing*, *serialising* and usually some limited reasoning support. With parsing we mean taking as input the triples serialised in different formats such as RDF/XML, Turtle, N3 etc. and representing and storing them internally as RDF triples, whereas, with serialising we mean taking the RDF triples in the triple store and exporting them in one of these mentioned formats. Going back to the query and update operations, triple stores typically use the standard SPARQL language. So-called "SPARQL-endpoints", that is, web services accessible via the standard SPARQL protocol, are able to delegate SPARQL requests over to a triple store.

SPARQL queries posed over a triple store—depending on the use case, or for the sake of query completeness—should also return implicit data that can be inferred by ontologies, hence such systems should have a means to exploit the ontology. In order to achieve this goal, different triple stores equipped with reasoning capabilities, based on the expressivity of the query and respectively of the ontology language, are able to perform *re-writing* of the query with respect to an ontology. This is known as the "top-down" approach to reasoning. Or, on the other hand, the alternative is to start from the triples and recursively derive all implicit triples with respect to ontology, until no new triples can be derived, and to store them explicitly in the triple store. This reasoning technique called *materialisation* (or reaching the "fix-point" in deductive databases), is also known as "bottom-up" approach. Both techniques have clearly advantages and disadvantages: materialisation would be more suitable in a data-warehouse context, as such it would improve the performance of query evaluation, but at the expense of re-materialisation due to the underlying data changes; whereas re-writing would be more suitable in the use cases where data changes are more frequent and queries are less time-critical. Also, in the context of more expressive ontology languages e.g., DL-Lite, it is not possible to do materialisation, since it allows for mandatory participations and cycles in the assertions resulting with an infinite chase. As mentioned earlier, for instance DBpedia performs (a partial, based on the hierarchy of classes) materialisation, falling into the category of bottom-up reasoning. SPARQL Entailment regimes [GOH+13] specify a semantics for SPARQL queries on how a SPARQL-endpoint should deal with queries respecting ontological inferences. However, for updates it is still an open issue on how a SPARQL endpoint should treat implicit triples, more specifically, how a delete, insert or update operation affecting an implicit triple should be handled in a standard way by such a system.

Apart from Linked Data and Semantic Web, in the context of data integration [Len02],

[12] `http://www.allegrograph.com`
[13] `https://www.blazegraph.com`

integrating data using ontologies is a topic which recently has been getting significant attention. In particular, in the data integration framework called *Ontology-Based Data Management* (OBDM) [Len18]—which extends *Ontology-Based Data Access* (OBDA) [PLC+08] so that not only queries, but also updates are considered—the original data are physically stored in legacy relational databases (as well as other standard heterogeneous formats such as XML, CSV, JSON[14]), which are exposed as a *virtual view* using a set of *mappings*[15] and an ontology vocabulary. A Description Logics knowledge base is used as both a global vocabulary and as a conceptual view (TBox), whereas the data reside in the original location, and are not extracted and physically materialised as in the case of triple stores. The user is able to query the view by using the vocabulary of the ontology, but she has no idea where and how the data is stored, which in turn can also be federated.

We mention three main reasons to use OBDM as opposed to the classic ETL and triple stores, and depending on these driving the respective use cases:

- *Synchronisation.* Data need not to be synchronised, as it is residing in the legacy, typically relational layer only. Hence, data is always up-to-date and no need for any additional data governance workflows to keep relational and ontology layers in sync;
- *Duplication.* Data is not duplicated, but instead only a view is computed in the ontology layer. This can be critical in the context of Big Data, such as stream data processing, where data storage is of crucial importance;
- *Metadata preservation.* The integrity constraints encoded in the relational layer are very often neglected and not consequently translated when an RDF "data lake" is created in the triple store. This calls for explicit translation to OWL axioms or SHACL[16] rules.

OBDM systems such as Ontop [CCK+17], MASTRO [CCD+13], Ultrawrap [Seq16], in the process of evaluating queries, usually employ three phases: first, the query by exploiting the ontology, is re-written in a top-down fashion by the system[17]; second, the rewritten query is translated back to a SQL query – a step called unfolding; and in the last step, the SQL query is executed over a relational database. The results are bindings of variables of the original query to resources or values. Queries can either be expressed as a conjunctive query (CQ) over the ontology or—as a syntactic variant thereof—as a SPARQL query.

[14] For instance, refer to [BCC+16] as regards to OBDM on top of MongoDB.

[15] Yet, another notion for wrappers.

[16] `https://www.w3.org/TR/shacl/`

[17] As a side note, Ontop perform the so-called "mapping saturation" instead of query rewriting to reach the very same result [CCK+17].

Nowadays, some triple store vendors such as Stardog, are going towards hybrid systems offering "virtual graph" functionalities, in this way mixing both materialised and virtual graphs, which support reasoning via query rewriting.

Regarding OBDM, even though there exists a well-established body of works for queries, the same unfortunately cannot be said for updates. The problem of updates in the context of OBDM is challenging since the problem lies in updating a database view, which still is an open problem in the area of database theory. The updates despite having to take into account the ontology—as stated previously in the case of queries—have also to be exactly translated back to the underlying relational sources. Such translation can be both ambiguous and it can also introduce facts (namely tuples) in the view, which are not always intuitive with respect to the original update, i.e., tuples known as "side-effects". It is of paramount importance, especially for enterprises, to have a means to correctly update the view, as such a functionality is very essential in practice. Typically, enterprises are reluctant to "change management", as such creating a (semantic) view on top of existing legacy data and providing read and write operations while preserving the old business processes and data workflows is often a more viable option for them.

1.2 Methodology

In this dissertation, we tackle the problem of updates in the context of both triple stores respecting ontological inference as well as OBDM, by de-constructing it into the core OBDM components. We start from a minimal setting, and then extend incrementally the expressivity by introducing ontology language or/and mapping constructs. This approach will enable us to see the problem through different lenses and thus to tackle the problem within specific settings: we approach the problem in a step-by-step fashion, i.e., starting with an ontology and data extracted physically in a triple store and not using mappings at all. Later on, we introduce new types of ontology axioms (constructs) and analyse the new challenges that arise. The same procedure is followed for adding mapping constructs as well. Likewise, we first consider only ABox updates, whereas afterwards we also discuss implications of TBox updates.

This dissertation tackles the original problem starting with minimal requirements, by using the minimal possible fragment, i.e., the minimal RDFS ontology language and no mappings at all[18]. We emphasize that, it is known that the existing structured data on the Web are currently using mostly RDFS, plus some few OWL axioms [GHKP12].

In the belief revision literature, there are in general two approaches dealing with updates: *formula-based* and *model-based* approaches [Win05]. In formula-based approaches the update and the knowledge base are represented as a set of axioms, and a result is a satisfiable conjunction of these two sets of axioms. In the model-based approaches, one is looking for a set of update models that in some specified way minimise the distance to

[18]In fact, one can easily translate the OBDM setting with no mappings to the respective one where 1-1 (one-to-one) mappings exist [CDGL$^+$07].

the set of original models. All the studies in the model-based area came up with solutions which are not satisfactory (the result of an update is not always expressible in the same logic [CKNZ10]), henceforth we focus on formula-based approaches to updates, which specifically we aim at implementing them by rewritings in terms of SPARQL updates, consistent with the standard semantics of SPARQL update and therefore implementable with "onboard" means of standard triple stores that support SPARQL.

Using the formula-based approach to updates and relying on implementable solutions, we tackle the problem in a "from theory to practice" methodology: by having covered the theory and all the corner cases that come along with the interplay of updates and reasoning (using the respective ontology fragments), it opens the way for concrete implementation. Thus, the problem is tackled by having a well-defined, unambiguous and formal *update semantics* that this dissertation strives to define and implementing it via *update rewritings in SPARQL*. As it is the case with belief revision, each of these semantics ought to be judged against a given set of *postulates* that should be fulfilled. Therefore, inspired by belief revision we have designed and adopted a special set of postulates for SPARQL/Update and compare each semantics against these postulates.

In particular, in this dissertation we are going to look into what we call *materialised-preserving* and *reduced-preserving* semantics. Materialised-preserving semantics always leaves the triple store in a materialised state. Reduced-preserving semantics, in the other extreme, would leave just the core of the ABox[19], and consequently not store any other facts which can be derived.

Once the update semantics are defined in terms of SPARQL rewritings, then one can test their feasibility by implementing the rewriting. For this, one can take any off-the-shelf triple store that implements the SPARQL specification and on top of that architecture develop the re-writing as defined by the respective update semantics. This dissertation also contributes a "benchmark" in the form of a set of updates based on the LUBM [GPH05] as ontology and benchmark for querying to perform experiments with different university sizes. The idea is to check how the different semantics perform in real-case scenarios and on different systems.

Besides increasing expressivity of ontological axiomatic constructs, the other major challenge is when increasing expressivity in the mapping language to the underlying (relational) data. In the case where mappings are not restricted to be one-to-one, we might end up with ambiguous update translations. In many-to-one mapping settings where two schema columns are mapped to the same entity/resource, then an insert operation is ambiguous. Also, in the case where joins are present, upon updates we end up in the same problematic cases that are known from the view update problem [Kel85][BS81][FG13][DB82].

In the following, we introduce the research problems that are tackled by this dissertation together with the corresponding challenges, contributions, and impact in terms of

[19] As we will see, the core is uniquely determined for the minimal fragment of RDFS, but not necessarily for other entailment regimes defined on top of more expressive ontology languages.

scientific publications.

1.3 Problem 1: Updating Implicit ABox and TBox Triples using SPARQL

The first problem we tackle in this dissertation is to define an update semantics that defines the behaviour of a SPARQL endpoint in a standard way when encountering the task of updating implicit triples using SPARQL. The input is a SPARQL update request. The main research question is the following:

What does it mean to *delete*, *insert*, as well as *update* an implicit triple?

We assume that the triples are either stored in a materialised or reduced triple store. Therefore, such semantics should also preserve materialisation or reducedness. Now, the input is a SPARQL update and a triple store which is already in a materialised (respectively, reduced) state. The more specific research question is the following:

Given a SPARQL update request, how can we preserve materialisation or reducedness of the triple store while capturing the intuition of the update? Which additional triples should be deleted or inserted in order to achieve this intuition?

The last part of the research question calls for an efficient update semantics, which ideally should be performed using rewritings instead of relying upon the costly materialise/reduce operator.

The first challenge is to define materialised- and reduced-preserving semantics for ABox updates in SPARQL. The second challenge is to also consider semantics for TBox updates in SPARQL. After that, we will study the proposed update semantics with respect to a set of postulates similar to belief revision.

Challenge 1.1: Updating Implicit RDFS ABox-es

The challenge is to update implicit ABox-es, i.e., instance triples in materialised or reduced triple stores. We are considering a triple store as container that contains RDFS axioms and assertions, namely, schema and instances, where either the materialise or reduce operator is already applied using the respective RDFS rules. In this setting, a triple can be derived from different triples in a derivation tree, due to four minimal RDFS axioms: subclass, subproperty, domain and range constraints. These axioms are used in order to perform materialisation or reducedness of the triple store. Current state-of-the-art approaches propose different semantics, using different tractable fragments of logic ranging from RDFS [GHV11] to DL-Lite [CKNZ10][LS11], albeit they do not consider general and bulk SPARQL updates. An operation for deleting implicit triples, calls for traversing back and deleting all triples in a derivation tree, so-called "causes". We restrict ourselves to the minimal RDFS fragment, as in this fragment it is feasible to compute such causes non-ambiguously and thus always achieve a deterministic result. This does

not hold in a more expressive ontology languages containing a disjunction operator in the left-hand side of such axiom constructs, i.e., in premises.

Contribution 1.1: Several Materialised-preserving ABox Update Semantics

This dissertation contributes with several materialised-preserving update semantics for SPARQL under minimal RDFS entailment (see Fig. 11): $\mathbf{Sem}_0^{mat}$, $\mathbf{Sem}_{1a}^{mat}$, $\mathbf{Sem}_{1b}^{mat}$, $\mathbf{Sem}_2^{mat}$ and $\mathbf{Sem}_3^{mat}$. $\mathbf{Sem}_0^{mat}$ is the baseline semantics that performs a naive update followed by the materialise operation. This semantics is commonly used in all the implementations of SPARQL endpoints, and is used as a baseline semantics to be compared with. $\mathbf{Sem}_{1a}^{mat}$ and $\mathbf{Sem}_{1b}^{mat}$ deal with the problem of deleting triples altogether with all the corresponding derived triples—so-called "dangling effects"—which are not derived by other non-deleted triples in any other alternative derivation path. $\mathbf{Sem}_{1b}^{mat}$ is a variant of $\mathbf{Sem}_{1a}^{mat}$ which distinguishes between explicit and implicit triples by keeping them stored separately. Both of these semantics rely on the materialise operator, making them more computationally intensive. $\mathbf{Sem}_2^{mat}$ deals with the problem of deleting implicit triples by deleting all the "causes" of triples that are designated to be deleted. This semantics is defined by a rewriting-based approach and not based on the materialise operator. $\mathbf{Sem}_3^{mat}$ can be thought as a combination of $\mathbf{Sem}_{1a}^{mat}$ or $\mathbf{Sem}_{1b}^{mat}$ with $\mathbf{Sem}_2^{mat}$. As follows from their definition, this semantics deletes all the causes of the triples designated to be deleted, as well as all of their dangling effects.
These update semantics are then characterised by a set of postulates, adapted from the well-known AGM framework [AGM85] and "translated" into SPARQL context.

Contribution 1.2: Two Reduced-Preserving ABox Update Semantics

This dissertation contributes with two reduced-preserving update semantics for SPARQL (see Fig. 11): $\mathbf{Sem}_0^{red}$ and $\mathbf{Sem}_1^{red}$. $\mathbf{Sem}_0^{red}$ is the baseline reduce semantics, similar to $\mathbf{Sem}_0^{mat}$, it performs a naive update followed by the reduce operator. $\mathbf{Sem}_1^{red}$ extends $\mathbf{Sem}_0^{red}$ by also deleting the causes of the triple designated to be deleted, as it is the case with $\mathbf{Sem}_2^{mat}$. As with the materialised-preserving semantics, both reduced-preserving semantics are characterised by a set of postulates inspired by the AGM framework.

Challenge 1.2: Updating Implicit RDFS TBox-es

The second main challenge is on updating TBox axioms, i.e., schema triples in materialised triple stores. In this case, materialisation has to take into account also RDFS TBox inference rules, which are used to do a transitive closure on subclass and subproperty axioms respectively. In fact, inserts are trivial as they merely boil down to a merge of graphs. On the other hand, deletes are challenging as there not always exists a deterministic way of performing a deletion of implicit schema triples. This holds even for deleting a single implicit schema triple. State-of-the-art approaches perform a minimal multicut on a graph, yielding a possible number of candidates for the minimal

multicut [GHV11]. The number of such combinations is exponential in the worst case with respect to the size of the schema. In practice it is more feasible as schemas tend to be smaller and instances are considered much bigger. This dissertation strives to define a semantics, which provides a canonical, namely deterministic way of performing a deletion of an implicit schema triple.

Contribution 1.3: Several Materialised-Preserving TBox Update Semantics

This dissertation contributes with two materialised-preserving semantics for TBox updates (see Fig. 11): $\mathbf{Sem}^{mat}_{incut}$ and $\mathbf{Sem}^{mat}_{outcut}$. We assume that the triple store is already in materialised state, i.e., transitive closure is applied. The idea behind $\mathbf{Sem}^{mat}_{outcut}$ is for every triple `:A rdfs:subClassOf :B` ought to be deleted, to delete all directly outgoing `rdfs:subClassOf` edges from A that lead into paths to B, or, resp., in $\mathbf{Sem}^{mat}_{incut}$ all directly incoming edges to B. As such, both semantics provide a canonical means of performing a deletion of an implicit TBox triple. They both rely upon a rewriting-based approach using SPARQL 1.1 property path queries. As with ABox semantics, the proposed TBox semantics are characterised in terms of postulates, inspired by the AGM framework.

Impact

The above contributions have resulted in the following peer-reviewed publications:

- Albin Ahmeti and Axel Polleres. *SPARQL update under RDFS entailment in fully materialized and redundancy-free triple stores.* In 2nd International Workshop on Ordering and Reasoning (OrdRing 2013), CEUR Workshop Proceedings, Sydney, Australia, October 2013. CEUR-WS.org.
- Albin Ahmeti, Diego Calvanese, and Axel Polleres. *Updating RDFS ABoxes and TBoxes in SPARQL.* In Proceedings of the 13th International Semantic Web Conference (ISWC 2014), Lecture Notes in Computer Science (LNCS). Springer, October 2014.
- Albin Ahmeti, Diego Calvanese, and Axel Polleres. *SPARQL Update for Materialized Triple Stores under DL-Lite$_{RDFS}$ Entailment.* In 27th International Workshop on Description Logics (DL2014), Vienna, Austria, July 2014.

1.4 Problem 2: Preserving Consistency and Materialisation in ABox Updates using SPARQL

The goal of the second problem is to define an update semantics, which defines the behaviour of a SPARQL endpoint in an exact, standard way when encountering the task of preserving the consistency of a triple store using SPARQL under entailment regimes

on top of minimal RDFS ontologies that additionally allow class disjointness axioms. We call this fragment RDFS$_\neg$. The input is a SPARQL update and a triple store in materialised state. The research question is the following:

Given a SPARQL update posed over a triple store, how can we preserve consistency as well as materialisation of the triple store? Which triples (belonging to the update vs triple store) should be given priority when resolving inconsistencies?

Given the fact that general SPARQL updates contain variables yet to be instantiated, such variables can lead to triples that violate the TBox without considering the ABox. We call such updates "intrinsically inconsistent". Note that this problem is not only typical of SPARQL, but also has surfaced in other OBDM approaches, e.g., [DLO$^+$17] that given the update at the source level and given the mappings, it aims to compute the corresponding update at the ontology level. Such update can easily be inconsistent because the rules at the ontology level are not necessarily specified at the source level. In this context, the first sub-challenge is to define a semantics for intrinsically inconsistent updates. The second sub-challenge is to define a consistency- and materialised-preserving update semantics for general ABox updates in SPARQL.

Challenge 2: Updating RDFS$_\neg$ ABox-es in materialised triple stores

The next challenge lies in resolving inconsistencies that occur due to newly inserted triples, which contradict with the old triples that are already in the triple store. The contradiction is now possible because of the addition of class disjointness axioms (negation $\neg$) to RDFS. As this problem is well-known in the area of belief revision, typically one has either to give priority to the new triples, old ones or just insert the subset of triples that are consistent. Nonetheless, this problem has not been studied in the context of SPARQL before. On top of that, as mentioned before, in SPARQL one can also encounter intrinsically inconsistent updates.

Contribution 2.1: Safe rewriting

This dissertation contributes with a semantics called "safe rewriting" that deals with the problem of intrinsically inconsistent updates. Safe rewriting removes all the inconsistent triples from the insert template of the SPARQL update. These triples are inconsistent in isolation with respect to the TBox alone, and not with respect to the ABox triples in the triple store. This approach of removing all "intrinsically" inconsistent triples is considered the safest approach, as there is no cue of which triples we shall keep and the ones we shall drop.

Contribution 2.2: Several consistency- and materalised-preserving ABox update semantics

On top of the initial, safe rewriting, this dissertation contributes with several consistency- and materialised-preserving ABox update semantics (see Fig. 11): brave $\mathbf{Sem}^{mat}_{brave}$, cau-

tious $\mathbf{Sem}^{mat}_{caut}$ and faint-hearted $\mathbf{Sem}^{mat}_{faint}$. Brave semantics $\mathbf{Sem}^{mat}_{brave}$ gives priority to the new triples to be inserted by removing the inconsistent triples (i.e., triples that lead to inconsistency with respect to TBox) residing in the triple store, whereas cautious $\mathbf{Sem}^{mat}_{caut}$ semantics drops entirely an update in case it is inconsistent with respect to triples residing in the triple store. Faint-hearted semantics $\mathbf{Sem}^{mat}_{faint}$ is in-between $\mathbf{Sem}^{mat}_{brave}$ and $\mathbf{Sem}^{mat}_{caut}$, by inserting only a consistent subset of triples. All three semantics are defined again in terms of SPARQL update rewritings. On top of that, all the semantics are characterised in terms of postulates, inspired by the AGM framework.

Impact

The above contributions resulted in the following peer-reviewed publications:

- Albin Ahmeti, Diego Calvanese, Vadim Savenkov, and Axel Polleres. *Dealing with Inconsistencies due to Class Disjointness in SPARQL Update.* In 28th International Workshop on Description Logics (DL2015), Athens, Greece, June 2015.
- Albin Ahmeti, Diego Calvanese, Axel Polleres, and Vadim Savenkov. *Handling inconsistencies due to class disjointness in SPARQL updates.* In Harald Sack, Eva Blomqvist, Mathieu d'Aquin, Chiara Ghidini, Simone Paolo Ponzetto, and Christoph Lange, editors, Proceedings of the 13th European Semantic Web Conference (ESWC2016), volume 9678 of Lecture Notes in Computer Science (LNCS), pages 387–404, Heraklion, Greece, June 2016. Springer.

1.5 Problem 3: Updating DBpedia using SPARQL

As the final piece of research, we looked into a real use-case of updates in OBDM, namely on the problem of updating DBpedia using SPARQL: In order to translate SPARQL updates posed over DBpedia, one has to propagate updates to Wikipedia infoboxes by resolving DBpedia mappings. The input is a SPARQL update and the DBpedia OBDM, i.e., OBDM consisting of DBpedia ontology, DBpedia mappings and Wikipedia infoboxes. The research question is the following:

> Given a SPARQL update, how can we resolve mappings and translate it into updates of the underlying DBpedia infoboxes? Or, the other way around, can any of the update semantics discussed before capture an infobox update?

The first challenge is on top-down[20] approach to updates in OBDM, where the focus is on the translation of SPARQL updates posed over DBpedia to Wiki updates. The second challenge is the bottom-up approach where the focus is to find an update semantics which captures the corresponding infobox update.

[20]To clarify here: top-down and bottom-up in this context refer to the "update translation", and not to "reasoning" as used before.

Challenge 3.1: Updates in OBDM - Top-down: Updating DBpedia triples using mappings and infoboxes

The first challenge is on updates in OBDM, i.e., the approach where the goal is to find an adequate translation of SPARQL updates to Wiki updates by resolving DBpedia mappings. This problem is similar to the view update problem in database theory, which in the general case has not been solved yet as of now. The mappings in the view update problem can comprise of join operations, whereas in DBpedia case, we can still have many-to-one, many-to-many mappings leading to ambiguity of translation as well. The current state-of-the-art approaches typically deal with either restricting mappings to one-to-one and considering the TBox fixed in the process [HRG10], or by providing user-interaction [NRG12] in order to get feedback from the (expert) users which option to choose from. We are rather interested in a solution that does not limit the expressivity of the mapping, but helps users to "break ties" with statistical information based on the real data used in practice. Therefore, having Wikipedia as a curated source with a history of updates captured through Wikipedia's update history, we have adopted DBpedia as a use case.

Challenge 3.2: Updates in OBDM - Bottom-up: Dealing with side-effects of wiki updates

The final challenge is on dealing with bottom-up translation of updates, i.e., given a Wiki infobox update as well as DBpedia infobox mappings, find the exact corresponding SPARQL update. On top of that, we have to include the DBpedia ontology (TBox) in the translation process as well. A Wiki infobox update can cause deletion or insertion of triples in DBpedia—due to non-monotonic DBpedia mappings—consequently potentially leading to introducing new inconsistent triples with respect to DBpedia TBox. Therefore, the update semantics we discussed in Problem 2 can be adopted to deal with this challenge.

Contribution 3: DBpedia-SUE

The last contribution of this dissertation is DBpedia-SUE (DBpedia SPARQL/Update Endpoint), an implementation of SPARQL updates for DBpedia which deals with the previous challenges. DBpedia-SUE takes a SPARQL update, evaluates the WHERE clause and processes the results in a triple-by-triple fashion. For each of the triples, it computes a corresponding set of wiki update translations. For insert operations, in case of ambiguous mappings (i.e., mappings that lead to more than one update translation) it gives a statistical information based on existing Wikipedia data on which infobox property is more probable as an adequate translation for the insertion. For delete (as well as insert) operations, it relies upon the previous user interactions: for each interaction stores them as patterns, and afterwards retrieves these patterns in future interactions – in this way assisting the user. The resulting infobox updates based on mappings can

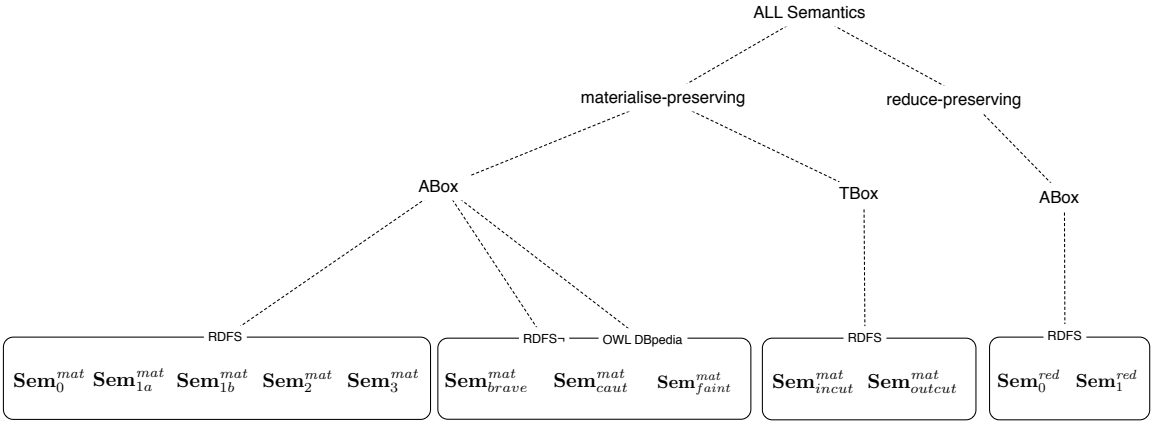


Figure 11: A tree representation of all the update semantics discussed in this dissertation grouped into materialise- and reduce-preserving.

re-trigger additional updates in DBpedia, which are treated as additional updates posed over DBpedia.

Impact

The above contribution resulted in the following peer-reviewed publications:

- Albin Ahmeti, Simon Razniewski and Axel Polleres. *Assessing the Completeness of Entities in Knowledge Bases.* ESWC2017 Poster & Demos, 2017.

- Albin Ahmeti, Javier Fernández, Axel Polleres, and Vadim Savenkov. *Updating wikipedia via DBpedia mappings and SPARQL.* In Eva Blomqvist, Diana Maynard, Aldo Gangemi, Rinke Hoekstra, Pascal Hitzler, and Olaf Hartig, editors, Proceedings of the 14th European Semantic Web Conference (ESWC2017), volume 10249 of Lecture Notes in Computer Science (LNCS), pages 485–501, Portoro, Slovenia, May 2017. Springer.

- Vadim Savenkov, Albin Ahmeti, Javier D. Fernández, and Axel Polleres. *Towards updating Wikipedia via DBpedia mappings and SPARQL.* In Alberto Mendelzon International Workshop on Foundations of Data Management (AMW2016), Panama City, Panama, June 2016. Short paper.

1.6 Structure of the Dissertation

This dissertation is structured as follows. The next Chapter 2 prepares the ground by introducing the theory and necessary definitions that are needed for the upcoming chapters. Chapter 3 defines and motivates the update problem, as well as it describes a set of postulates to be fulfilled by the proposed update semantics discussed in the next chapters. Chapter 4 is about updating implicit ABox and TBox triples using SPARQL, where several update semantics are introduced. Chapter 5 is about preserving consistency in ABox updates using SPARQL, where three update semantics are introduced. Chapter 6 is about the real use case of updates in OBDM: Updates in DBpedia. Chapter 7 discusses the implementation and experimental evaluations of previous three chapters. In the end, Chapter 8 puts the dissertation's work into the context of the state-of-the-art, reflecting upon the novelties provided by the dissertation, and finally commenting on the issues that are left open beyond the scope of the dissertation for future work.

CHAPTER 2

Preliminaries

In this chapter, we provide all the basic definitions that are used throughout the next chapters of this dissertation. Other definitions that are specific to a chapter will be introduced in the respective chapter as it goes on. Initially, we define a more abstract, general setting of Ontology-Based Data Management, which we then put it into a more tangible Semantic Web setting.

2.1 Ontology-Based Data Management

Ontology-Based Data Management [PLC+08] [Len18] [XCK+18] is a data-integration framework, which defines a fixed, common ontology connecting relational databases via a set of mappings. The ontology is used to model a domain at the conceptual level, encoding the semantics of the application by providing reasoning capabilities as well. Between the relational and ontology models—that each form a different layer—there is an impedance mismatch: integrity constraints in the relational databases, such as primary and foreign key constraints are mandatory, whereas in the ontology layer not necessarily; relational tables in databases consist of arbitrary n-tuples, whereas ABox assertions in ontologies refer to binary relations at most. Ontologies are rather meant to model and deal with incomplete information adhering to the Open World Assumption (OWA) as opposed to the Closed World Assumption (CWA) in databases. With OWA a fact is inferred to be unknown if it is not known to be true, whereas with CWA the opposite holds, i.e., a fact is inferred to be false if it is not known to be true. This mismatch between these models is addressed by means of mappings, which are in principle sets of rules meant to bridge these two worlds. In the OBDM setting, data is not physically stored in the ontology layer, hence queries defined in terms of the ontology concepts and roles are translated to the database layer.

Users interacting with OBDM via a query language and pose queries to the ontology layer, which are automatically translated to the underlying databases by query rewriting

techniques. In fact, OBDM systems typically use three-step operations: query rewriting, query unfolding and query execution. Hence, a lot of details are hidden and abstractions are transparent to the user. In the following the necessary definitions are given on OBDM, starting from the bottom, i.e., the relational layer. We give only the basic definitions of relational databases, the reader is referred to [RG03] for more details.

Definition 1 (Schemas, Instances) *A* database schema $\mathcal{S}$ *is a finite set of relation schemas* $\mathcal{S} = \{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$*, where* $\mathcal{R}_i$ *consists of a finite set of attributes which we denote by [] and whose size* m *is called* the arity of $\mathcal{R}_i$*. Each attribute as a domain has a fixed, countably infinite set* Δ *of constants. An* instance $\mathcal{D}$ *of* $\mathcal{S}$ *consists of a relation* R_i*, for every relation schema* $\mathcal{R}_i$*, i.e.,* $\mathcal{D} = \{R_1, \ldots, R_n\}$*, where relation* R_i *is a set of tuples* $a = (a_1, \ldots, a_m)$*, such that* m *is the arity of* R_i*, by mapping each attribute of* R_i *to a value in* Δ *or to the* `NULL` *value. The active domain* $\mathrm{adom}(\mathcal{D})$ *of* $\mathcal{D}$ *is the set of all constants and* `NULL` *that occur in the tuples of* $\mathcal{D}$*.*

Example 1 Let $\mathcal{S}$ be the schema consisting of relations with the following signature:

employees[`ID:`**`INTEGER`**`, EMPNAME:`**`VARCHAR`**`, City:`**`VARCHAR`**`, Salary:`**`VARCHAR`**],
departments[`ID:`**`INTEGER`**`, DEPTNAME:`**`VARCHAR`**],
empdepts[`EMPID:`**`INTEGER`**`, DEPTID:`**`INTEGER`**],

and let the database instance $\mathcal{D}$ consist of the tuples shown in Fig. 21:

employees = {(1, John, Vienna, 2400), (2, Joe, London, 2300), (3, Anna, Rome, 2700), ...}
departments = {(101, Finance), (102, Marketing), ...}
empdepts = {(1, 101), (2, 102), (3, 101), (3, 102), ...}

■

Definition 2 (Database queries) *A query over* $\mathcal{S}$ *is semantically defined as a mapping* q *that associates with every instance* $\mathcal{D}$ *of* $\mathcal{S}$ *a set of answers* $q(\mathcal{D}) \subseteq adom(\mathcal{D})^n$*, where* $n \geqslant 0$ *is the arity of* q*. If* $n = 0$*, then we say that* q *is a Boolean query, and we write* $q(\mathcal{D}) = \top$ *if* $() \in q(\mathcal{D})$ *and* $q(\mathcal{D}) = \bot$ *otherwise. Queries can be specified by means of a first-order formula* $\phi = f(x_1, \ldots, x_n)$ *with free variables* $x_1, \ldots, x_n$*, also often referred to as a FO-query, that uses only relation names from* $\mathcal{S}$ *(and, possibly,* $=, <, >$*). Then query evaluation comes down to checking satisfaction in first-order logic:*

$$q_{\phi,\mathcal{S}}(\mathcal{D}) = \{(a_1, \ldots, a_n) \in adom(\mathcal{D})^n \mid \mathcal{D} \models \phi[a_1, \ldots, a_n]\}$$

Here, $\phi[a_1, \ldots, a_n] = f(x_1/a_1, \ldots, x_n/a_n)$*, i.e., a substitution of variables* x_i *with actual values* a_i*. Furthermore, instead of FO-queries we will often refer to SQL queries that are common in databases*[1]*.*

[1] As a remark, we mention that the semantics of SQL is based on bags as opposed to sets in FOL. However, herein we consider FO-expressible SQL queries with set semantics.

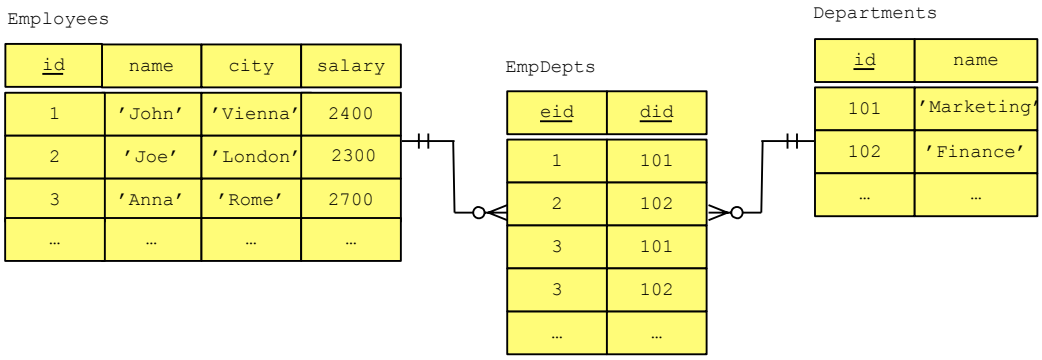


Figure 21: The figure depicts an adopted fragment of the classic "employee-department" database schema populated with hypothetical data.

Definition 3 (Integrity constraints) *There are two main types of integrity constraints applied to schemas $\mathcal{S}$:*

- Functional dependencies *(e.g., primary key constraints): "$X \rightarrow A$", where X is a set of attributes and A an attribute. Intuition: The value of A is fully determined by the values of X. As a first-order formula we can express functional dependencies:* $\forall \vec{x}\vec{y}\vec{z}uv(R(\vec{x},\vec{y},u) \wedge R(\vec{x},\vec{z},v) \rightarrow u = v)$. *We define* candidate key *as a minimal set of attributes that functionally determine all of the attributes in a relation. A* primary key *is a choice of candidate key. We denote the primary keys of a relation R with* $\text{key}(R) = X$.

- Inclusion dependencies *(e.g., foreign key constraints): "$\mathcal{R}[X] \subseteq \mathcal{S}[Y]$", where X is a subset of attributes in $\mathcal{R}$ and Y is a subset of attributes in $\mathcal{S}$ with the same arity. Intuition: All value combinations occurring in X also occur in Y. As a first-order formula we can express inclusion dependencies as:* $\forall \vec{x}\vec{y}(R(\vec{x},\vec{y}) \rightarrow (\exists \vec{z})S(\vec{x},\vec{z}))$

Example 2 Let the following integrity constraints be applied to schema $\mathcal{S}$ from Ex. 1:

$$key(employee) = \{ID\}$$
$$key(department) = \{ID\}$$
$$empdept[EMPID] \subseteq employee[ID]$$
$$empdept[DEPTID] \subseteq department[ID]$$

■

Definition 4 (Ontology) *An ontology $\mathcal{O}$ consists of first-order axioms represented in an ontology language. An ontology language $\mathcal{L}$ is a fragment of first-order logic (i.e., a set of FO axioms), and an $\mathcal{L}$-ontology $\mathcal{O}$ is a finite set of axioms from $\mathcal{L}$. In our context, we will be talking exclusively about axioms of predicates of arity one (representing concept membership) and two (representing role membership[2]). To define the semantics of an ontology, we rely on the standard notions of (first-order logic)* interpretation*,* satisfaction *of assertions, and* model*.*

Usually when speaking about ontologies, facts about instances denoting class and role membership (ABox) and schema axioms (TBox) are often distinguished. Specifically in Description Logics, ABox and TBox are asserted by means of relations of arity one and two, called concepts and roles respectively. In the following, we talk about the basic Description Logic $\mathcal{ALC}$, whereas throughout the dissertation we will adjust these concepts as needed in order to capture the expressivity of more expressive logics[3]. An $\mathcal{ALC}$-concept is formed according to the syntax rule:

$$C, D ::= A \mid \top \mid \bot \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \exists P.C \mid \forall P.C,$$

where A is an (atomic) concept name and P is a role name. We also refer to the signature or vocabulary Γ, which is the set of all individual, concept and role names.

An $\mathcal{ALC}$ TBox is a finite set of concept inclusions $C \sqsubseteq D$, with C and D being $\mathcal{ALC}$ concepts, whereas an ABox is a finite set of concept assertions $C(a)$ and role assertions $P(a, b)$. Together both $\mathcal{ALC}$ TBox axioms and ABox assertions comprise the $\mathcal{ALC}$ ontology. $\mathcal{ALCI}$ is an extension of $\mathcal{ALC}$ by capturing also role inverses.

Example 3 An example of an $\mathcal{ALCI}$ ontology $\mathcal{O}$ with TBox $\mathcal{T}$, is defined as follows (we will sometimes use UML notation to visualise ontological concepts, cf. Fig. 22):

[2]For simplicity, attributes (data properties) are treated in the following as they were to be roles (object properties).

[3]For more details refer to [BCM+03].

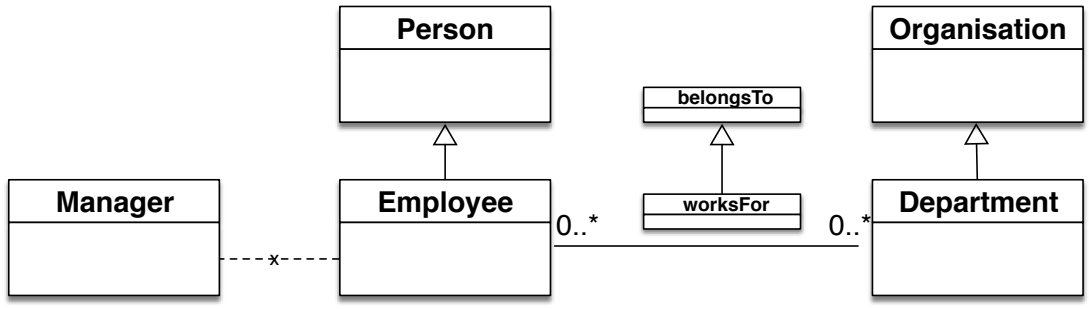


Figure 22: The figure conceptualises the axioms in Ex. 3 by using an UML diagram. For the sake of simplicity, we abuse the notation and for "subproperty" we use the same notation as for the UML "subclass". Notice that we also use the symbol 'x' to denote disjointness, which is not standard in UML.

$$
\begin{array}{rcl}
\exists worksFor.\top \sqsubseteq & Employee & \forall x, y.worksFor(x,y) \rightarrow Employee(x) \\
\exists worksFor^{-}.\top \sqsubseteq & Department & \forall x, y.worksFor(x,y) \rightarrow Department(y) \\
worksFor \sqsubseteq & belongsTo & \forall x, y.worksFor(x,y) \rightarrow belongsTo(x,y) \\
Employee \sqsubseteq & Person & \forall x.Employee(x) \rightarrow Person(x) \\
Department \sqsubseteq & Organization & \forall x.Department(x) \rightarrow Organization(x) \\
Manager \sqsubseteq & \neg Employee & \forall x.Manager(x) \rightarrow \neg Employee(x)
\end{array}
\quad \Longleftrightarrow
$$

$$
\begin{aligned}
\mathcal{A} = \{&worksFor(john, marketing), worksFor(joe, finance), \\
&worksFor(anna, marketing), worksFor(anna, finance)\}
\end{aligned}
$$

■

In the future, we will write axioms $\exists worksFor.\top \sqsubseteq Employee$ ($\exists worksFor^{-}.\top \sqsubseteq Department$) shortly as $\exists worksFor \sqsubseteq Employee$ ($\exists worksFor^{-} \sqsubseteq Department$) respectively. The semantics of $\mathcal{ALCI}$ is given via a translation to FO-axioms with one free variable as shown in the following [BtCLW13].

Definition 5 (Interpretation, satisfaction, model) *An $\mathcal{ALCI}$* interpretation $\langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$ *consists of a non-empty set $\Delta^{\mathcal{I}}$ called the object domain, and an interpretation function $\cdot^{\mathcal{I}}$, which maps:*

- *each atomic concept A to a subset $A^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$,*
- *each atomic role P to a binary relation $P^{\mathcal{I}}$ over $\Delta^{\mathcal{I}}$, and*
- *each element of Γ to an element of $\Delta^{\mathcal{I}}$.*

For concept expressions, the interpretation function is defined as follows:

- $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
- $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
- $(\exists P.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y.(x,y) \in P^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
- $(\forall P.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y.(x,y) \in P^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
- $(\exists P^{-}.C)^{\mathcal{I}} = \{y \in \Delta^{\mathcal{I}} \mid \exists x.(x,y) \in P^{\mathcal{I}} \wedge x \in C^{\mathcal{I}}\}$
- $(\forall P^{-}.C)^{\mathcal{I}} = \{y \in \Delta^{\mathcal{I}} \mid \forall x.(x,y) \in P^{\mathcal{I}} \rightarrow x \in C^{\mathcal{I}}\}$

An interpretation $\mathcal{I}$ satisfies *an inclusion assertion $C \sqsubseteq D$, if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Analogously, $\mathcal{I}$* satisfies *an ABox assertion of the form (where $x, y \in \Gamma$)*

- *$A(x)$, if $x^{\mathcal{I}} \in A^{\mathcal{I}}$, and*
- *$P(x,y)$, if $(x^{\mathcal{I}}, y^{\mathcal{I}}) \in P^{\mathcal{I}}$.*

An interpretation $\mathcal{I}$ is called a model *of an ontology $\mathcal{O}$ (resp., a TBox $\mathcal{T}$, an ABox $\mathcal{A}$), denoted $\mathcal{I} \models \mathcal{O}$ (resp., $\mathcal{I} \models \mathcal{T}$, $\mathcal{I} \models \mathcal{A}$), if $\mathcal{I}$ satisfies all assertions in $\mathcal{O}$ (resp., $\mathcal{T}$, $\mathcal{A}$).*

Definition 6 (Ontology queries) *In this dissertation, besides general FO-queries as defined in Def. 2, we also consider the special case of conjunctive queries posed over ontologies.*

A conjunctive query *(CQ) q is a rule of the form*

$$q(\vec{x}_0) \leftarrow body$$

where the body is a set of atoms of the form $A(x_1)$ and $P(x_1, x_2)$, and equalities; where x_i are terms, i.e., variables or constants. The variables in $\vec{x}_0$ are the distinguished variables, the others occurring in the query body only are the non-distinguished variables. Furthermore, $\vec{x}_0$ consists of distinct variables only. A union of conjunctive queries *(UCQ) Q is a set of CQs with the same arity.*

Example 4 The following conjunctive query asks for *all persons who work for the department of "finance"*:

$$q: \quad q(x) \leftarrow Person(x), worksFor(x,y), Department(y), y = finance$$

Note that x is a distinguished variable, whereas y is a non-distinguished variable. ■

Definition 7 (Certain answers) *For a CQ q with distinguished variables $\vec{x_0} = x_1, \ldots, x_n$ and an ontology $\mathcal{O}$, we say that $\vec{a} = a_1, \ldots, a_n$ is an answer for q in $\mathcal{O}$, in symbols $\mathcal{O} \models q[\vec{a}]$, if $\mathcal{I} \models q[\vec{a}]$ for every model $\mathcal{I}$ of $\mathcal{O}$. As defined before, $q[\vec{a}]$ is the substitution of distinguished variables $\vec{x_0}$ with values $\vec{a}$. By $ans(q, \mathcal{O})$ we denote the set of all answers for q in $\mathcal{O}$.*

Example 5 Given the previous query q in Ex. 4, TBox $\mathcal{T}$ and ABox $\mathcal{A}$ as defined in Ex. 3, then certain answers to the query q are:

$$ans(q, \mathcal{O}) = \{joe, anna\}$$

■

OBDM systems typically rewrite queries in order to return certain answers, as it will be discussed in the following. The most well-known query rewriting algorithm in the context of OBDM is called *PerfectRef*, which is tailored to ontologies restricted to the DL-Lite family of logics [CDGL$^+$07]. Given that in this dissertation we will leverage on *PerfectRef*, albeit it is used in the context of less expressive ontology languages, we explain herein DL-Lite in order to understand the difference of applying *PerfectRef* in various ontology languages. We will also discuss some of the related (state-of-the-art) approaches in Chapter 8 that are based on different fragments of the DL-Lite family of logics [ACKZ14]. DL-Lite is a family of logics able to express and capture most of the constructs of Entity-Relationship and UML diagrams, while at the same time keeping the computational complexity of reasoning tasks (most importantly, query answering) tractable. DL-Lite is able to capture the following constructs [CDGL$^+$07]:

- *ISA*, $A_1 \sqsubseteq A_2$,
- *disjointness*, $A_1 \sqsubseteq \neg A_2$,
- *role-typing*, $\exists P \sqsubseteq A$ ($\exists P^- \sqsubseteq A$),
- *mandatory participation*, $A \sqsubseteq \exists P$ ($A \sqsubseteq \exists P^-$),
- *mandatory non-participation*, $A \sqsubseteq \neg\exists P$ ($A \sqsubseteq \neg\exists P^-$),
- *functionality restrictions on roles*, $funct\ P$ ($funct\ P^-$).

DL-Lite provides a robust foundation for OBDM, as it is able to capture very important constructs, and query answering can be done via query rewriting that returns certain answers. Let us provide more details on the algorithm that computes the query rewriting.

PerfectRef rewrites the query q to q', which is computed starting from the UCQ $Q = q$ and expanding Q by exhaustively applying, to every CQ in Q the following rewriting steps:

1. *AtomRewrite* (lines 5-11, cf. Alg. 2.1): use every (positive[4]) inclusion axiom as a rewriting rule (from right to left);
2. *Reduce* (lines 12-16, cf. Alg. 2.1): apply the most general unifier[5] to two unifiable atoms.

[4]On the other hand, negative inclusion axioms $A_1 \sqsubseteq \neg A_2$ are not used in query answering, though they can be used to chase inconsistencies.

[5]Quoting Russell & Norvig [RN10], the most general unifier (or MGU) is the substitution that makes the least commitment on the bindings of the variables.

Algorithm 2.1: *rewrite*$(q, \mathcal{T})$

Input: Conjunctive query q, TBox $\mathcal{T}$
Output: Union (set) of conjunctive queries

```
1  P := {q}
2  repeat
3  |  P' := P
4  |  foreach q ∈ P' do
5  |  |  foreach g in q do        // expansion
6  |  |  |  foreach inclusion assertion I in T do
7  |  |  |  |  if I is applicable to g then
8  |  |  |  |  |  P := P ∪ {q[g/ gr(g, I)]}
9  |  |  |  |  end
10 |  |  |  end
11 |  |  end
12 |  |  foreach g1, g2 in q do     // reduction
13 |  |  |  if g1 and g2 unify then
14 |  |  |  |  P := P ∪ {τ(red(q, g1, g2))}
15 |  |  |  end
16 |  |  end
17 |  end
18 until P' = P
19 return P
```

Let us dissect the algorithm in more detail. *AtomRewrite* in its most basic form, uses (positive) inclusion axioms $C \sqsubseteq D$ to derive $C(x)$ from $D(x)$. In this way, we say that a TBox axiom $C \sqsubseteq D$ is *applicable* (line 7) to the atom $D(a)$, given that we do not have $C(a)$ yet on our list of processed atoms[6]. Arguments are not affected by the rewriting, they are merely propagated, i.e., preserving constants or variables. $gr(g, I)$ indicates the atom obtained from the atom g by applying the applicable inclusion axiom I. The semantics of $gr(g, I)$ is given in Table 21. In line 8, g is substituted by $gr(g, I)$, and $q[g/\,\mathrm{gr}(g, I)]$ denotes the conjunctive query obtained from q by replacing an atom g with a new atom $gr(g, I)$.

Reduce is a function (line 14) that takes as input a conjunctive query q and two atoms g_1 and g_2 occurring in the body of q, and returns a conjunctive query q' obtained by applying to q the most general unifier between g_1 and g_2. τ is a function that takes as input a conjunctive query q and returns a new conjunctive query obtained by replacing each occurrence of a non-distinguished variable in q which occurs only once with the symbol _, which stands as a placeholder for a "fresh", "anonymous" variable.

[6]Similarly, applying such axioms in the reverse direction can be used to create the *chase* or *materialisation* of the knowledge base.

g	I	$\mathrm{gr}(g/I)$
$A(x)$	$A_1 \sqsubseteq A$	$A_1(x)$
$A(x)$	$\exists P \sqsubseteq A$	$P(x, \_)$
$A(x)$	$\exists P^- \sqsubseteq A$	$P(\_, x)$
$P(x, y)$	$P_1 \sqsubseteq P$	$P_1(x, y)$
$P(x, \_)$	$A \sqsubseteq \exists P$	$A(x)$
$P(x, \_)$	$\exists P_1 \sqsubseteq \exists P$	$P_1(x, \_)$
$P(x, \_)$	$\exists P_1^- \sqsubseteq \exists P$	$P_1(\_, x)$
$P(\_, x)$	$A \sqsubseteq \exists P^-$	$A(x)$
$P(\_, x)$	$\exists P_1 \sqsubseteq \exists P^-$	$P_1(x, \_)$
$P(\_, x)$	$\exists P_1^- \sqsubseteq \exists P^-$	$P_1(\_, x)$
$P(x, y)$	$P_1 \sqsubseteq P^-$	$P_1(y, x)$

Table 21: Semantics of $\mathrm{gr}(g, I)$ in Alg. 2.1. In an atom, '_' stands for a "fresh" variable. Note that for the minimal RDFS rules [MPG07] only the first top-most four apply, whereas for DL-Lite all of them apply.

The following example shows the peculiarities of query rewriting under DL-Lite when both *AtomRewrite* and *Reduce* are applied, thus ensuring that the rewritten query indeed returns certain answers.

Example 6 (*PerfectRef* in DL-Lite) Consider an ontology with the following TBox and ABox:

$$\mathcal{T} = \{Employee \sqsubseteq \exists worksFor, worksFor \sqsubseteq belongsTo\}$$
$$\mathcal{A} = \{Employee(john)\}$$

and the query

$$q(x, y) \leftarrow worksFor(x, z), belongsTo(y, z) \tag{2.1}$$

First, an *AtomRewrite* step rewrites $belongsTo(y, z)$, another new CQ is generated:

$$q'(x, y) \leftarrow worksFor(x, z), worksFor(y, z) \tag{2.2}$$

Then, a *Reduce* step can be applied, and another new CQ is generated:

$$q'(x, x) \leftarrow worksFor(x, z) \tag{2.3}$$

which in turn is equivalent (after application of τ) with:

$$q'(x, x) \leftarrow worksFor(x, \_) \tag{2.4}$$

Now, after the *Reduce* step, again *AtomRewrite* can be applied and consequently using $Employee \sqsubseteq \exists worksFor$ another new CQ is generated:

$$q'(x, x) \leftarrow Employee(x) \tag{2.5}$$

The result of the rewriting is the union of the generated CQs plus the initial CQ ((2.1), (2.2), (2.4) and (2.5)). After the evaluation, the result of the rewritten query is: $john, john$. ■

In the end, let us stress that for each atom, *AtomRewrite* generates at most a linear number of rewritings with respect to the TBox size. Nonetheless, in the end of the rewriting process, the result is a UCQ that could potentially have an exponential number of CQs with respect to the number of atoms of the initial query. For this reason, there exist other query rewriting algorithms, to name a few *Presto* [RA10], *Rapid* [CTS11], *REQUIEM* [PMH09] etc., which focus on optimising, that is, reducing the number of queries generated by *PerfectRef*.

Next, let us switch the discussion to updates.

Definition 8 (General update operation) *Let ϕ_w be an FO-query (possibly involving disjunction). Then a general update operation $u(\phi_d, \phi_i, \phi_w)$ has the form:*

$$\textbf{\textit{DELETE}}\ \ \phi_d\ \ \textbf{\textit{INSERT}}\ \ \phi_i\ \ \textbf{\textit{WHERE}}\ \ \phi_w$$

Intuitively, given an ontology $\mathcal{O}$ and an update $u(\phi_d, \phi_i, \phi_w)$, where ϕ_d and ϕ_i are sets of:

- *ABox atoms over the ontology predicates with constants or variables from the free variables in the query ϕ_w (ABox update), or*
- *TBox axioms, where the allowed form varies with respect to the ontology language, i.e., for instance inclusion axioms, where variables from the free variables in the query ϕ_w can take the positions of ontology predicates (that is, typically roles or concepts) (TBox update).*

Then, the semantics of the general update operation is defined as $(\mathcal{O} \setminus \mathcal{O}_d) \cup \mathcal{O}_i$, where $\mathcal{O}_d = \bigcup_{\theta \in ans(\phi_w, \mathcal{O})} \phi_d\theta$ and $\mathcal{O}_i = \bigcup_{\theta \in ans(\phi_w, \mathcal{O})} \phi_i\theta$. Here, by θ we mean a substitution replacing the free variables according to answers of ϕ_w.

We note that in languages such as RDF where both assertional and terminological knowledge are expressed by triples, the borders between TBox and ABox updates are blurred and additional syntactic constraints are necessary to separate them, or, respectively, to avoid such updates to not create non-standard use [PHDU13] of the RDF(S) and OWL vocabulary.

Proposition 1 (Atomic update) *Given an ontology $\mathcal{O}$ and a general update $\mathcal{U} = u(\mathcal{O}_d, \mathcal{O}_i, \emptyset)$, where $\mathcal{O}_d$ ($\mathcal{O}_i$) are the facts and assertions to be removed (added respectively), then the semantics of the* atomic update *$\mathcal{U}$ over ontology $\mathcal{O}$ is defined simply as $\mathcal{U}(\mathcal{O}) = (\mathcal{O} \setminus \mathcal{O}_d) \cup \mathcal{O}_i$.*

Example 7 Consider the following general update u_1 that deletes and inserts—yet to be instantiated by a WHERE clause—ABox assertions of class *Person* and *Department* respectively:

```
DELETE { Person(x) }
INSERT { Department(y)}
WHERE { Person(x), worksFor(x, y), Department(y), y=finance}
```

after instantiation of the WHERE clause as in Ex. 5, it boils down to the following atomic update[7]:

```
DELETE { Person(joe), Person(anna)}
INSERT { Department(finance) }
```

Whereas this update affects the ABox, the following update operation u_2 = **INSERT** $\{ \forall x.Manager(x) \Rightarrow Person(x)\}$ incorporates a new TBox axiom to the ontology. ■

Notice that a deletion or insertion of TBox axioms based on a WHERE clause is not possible in FOL, because variables can not take the positions of concept or role names[8].

Now, we extend the definitions to the setting of OBDM. Before giving the semantics of OBDM, we initially give the definition of the syntax and semantics of mappings to the underlying relational data sources.

In order to create objects of the ontology from the tuples in the database we need constructors, i.e., speaking in formal terms we need Skolem functions. First, let us introduce an alphabet λ of function symbols f each associated with arity n, whose attributes are universally quantified variables occurring in the left-hand side of the mapping, as a means to produce distinct fresh values in the ontology, which depend on certain values in the source database.

[7] Note that this particular update has no effect, since it deletes and inserts implicit facts. In the future, we will see various update semantics that tackle this issue.

[8] This can be expressed in SPARQL though, for instance:

```
INSERT { ?s rdfs:subClassOf ?o } WHERE { ?s skos:broader ?o }.
```

When instantiated, a Skolem term $f(x_1, \ldots, x_n)$ gives rise to an *object term* $f(a_1, \ldots, a_n)$ where a_i is an instantiation of a variable x_i, for all i in $1 \ldots n$. We assume that such object terms are distinct from any constant occurring in a source database or in the target ontology, and thus each triggering of a rule with a distinct assignment $(a_1, \ldots, a_n)$ for the universally quantified values generates a distinct fresh value $f(a_1, \ldots, a_n)$ in the ontology.

Definition 9 (Mappings) *Let $\phi(\vec{x})$ be a query over $\mathcal{D}$, whereas $\psi(f(\vec{x}), \vec{x})$ is the body of the conjunctive query with distinguished variables, whose variables appearing in object terms $f(\vec{x})$ are from $\vec{x}$. Then, we define a* mapping *[Len02] between $\phi(\vec{x})$ and $\psi(f(\vec{x}), \vec{x})$ as:*

$$\phi(\vec{x}) \rightarrow \psi(f(\vec{x}), \vec{x})$$

In Ex. 9 examples of mappings in different formalisms, are given first by using the notation we just described and their counterpart using FOL and Datalog notation.

We proceed with the semantics of mappings.

Definition 10 (Semantics of mapping(s)) *We extend the interpretation, so that an interpretation $\mathcal{I}$ satisfies $\phi(\vec{x}) \rightarrow \psi(f(\vec{x}), \vec{x})$ w.r.t. a database instance $\mathcal{D}$, if for every tuple of values $\vec{v}$ in the answer of the query $\phi(\vec{x})$ over $\mathcal{D}$, and for each ground atom X in $\psi(f(\vec{v}), \vec{v})$, we have that:*

- *if X has the form $A(s)$, then $s^{\mathcal{I}} \in A^{\mathcal{I}}$*
- *if X has the form $P(s_1, s_2)$, then $(s_1^{\mathcal{I}}, s_2^{\mathcal{I}}) \in P^{\mathcal{I}}$.*

Example 8 Given $\mathcal{D}$ as defined in Ex. 1, mappings $\mathcal{M}$ as defined in Ex. 9, then we can compute the virtual view:

$$\begin{aligned}\mathcal{M}(\mathcal{D}) = \mathcal{A} \cup \{&Employee(john), name(john, John), Employee(joe),\\ &name(joe, Joe), Manager(anna), name(anna, Anna),\\ &Department(finance), deptName(finance, Finance),\\ &Department(marketing), deptName(marketing, Marketing),\\ &\ldots\}\end{aligned}$$

Note that for the sake of readability, we omit the object terms and we use the assertions as in Ex. 3, for instance, instead of $emp(3)$ we use *anna* and so on. ∎

Example 9 Examples of mappings $\mathcal{M}$ defined in different formalisms, starting from the top: FOL, Datalog and SQL respectively. Note that we use two different Skolem functions, namely $emp(\vec{x})$ and $dept(\vec{y})$:

Formalism	Mappings
FOL	$m_1 : \forall \vec{x}\vec{y}.\ empdept(\vec{x},\vec{y}) \rightarrow worksFor(emp(\vec{x}), dept(\vec{y}))$
	$m_2 : \forall \vec{x}\vec{y}.\exists zw.\ employee(\vec{x},\vec{y},z),\ empdept(\vec{x},w), z < 2500 \rightarrow Employee(emp(\vec{x})), name(emp(\vec{x}),\vec{y})$
	$m_3 : \forall \vec{x}\vec{y}.\exists zw.\ employee(\vec{x},\vec{y},z), empdept(\vec{x},w), z \geq 2500 \rightarrow Manager(emp(\vec{x})), name(emp(\vec{x}),\vec{y})$
	$m_4 : \forall \vec{x}\vec{y}.\exists z.\ department(\vec{x},\vec{y}), empdept(z,\vec{x}) \rightarrow Department(dept(\vec{x})), deptName(dept(\vec{x}),\vec{y})$
Datalog	m_1 : `worksFor(emp(X), dept(Y)) :- empdept(X, Y) .`
	m_2 : `Employee(emp(X)), name(emp(X), Y) :- employee(X, Y, Z), empdept(X, W), Z < 2500 .`
	m_3 : `Manager(emp(X)), name(emp(X), Y) :- employee(X, Y, Z), empdept(X, W), Z >= 2500 .`
	m_4 : `Department(dept(X)), deptName(dept(X), Y) :- department(X, Y), empdept(Z, X) .`
SQL	m_1 : **SELECT** EMPID, DEPTID **FROM** empdept $\rightarrow$ $worksFor(\mathbf{emp}(EMPID), \mathbf{dept}(DEPTID))$
	m_2 : **SELECT** ID, EmpName **FROM** employee emp, empdept ed **WHERE** emp.ID=empdept.EMPID **AND** emp.Salary < 2500 $\rightarrow$ $Employee(\mathbf{emp}(ID)), name(\mathbf{emp}(ID), EmpName)$
	m_3 : **SELECT** ID, EmpName **FROM** employee emp, empdept ed **WHERE** emp.ID=empdept.EMPID **AND** emp.Salary >= 2500 $\rightarrow$ $Manager(\mathbf{emp}(ID)), name(\mathbf{emp}(ID), EmpName)$
	m_4 : **SELECT** ID, DeptName **FROM** department dept, empdept ed **WHERE** dept.ID=empdept.DEPTID $\rightarrow$ $Department(\mathbf{dept}(ID)), deptName(\mathbf{dept}(ID), DeptName)$

■

OBDM is a framework which consists of *database schemas* $\mathcal{S}$, *mappings* $\mathcal{M}$, a common *ontology* $\mathcal{T}$, and concerns answering queries $\mathcal{Q}$ and updates $\mathcal{U}$ posed over $\mathcal{T}$. Each query and update language has its own syntax and respectively its own semantics. We formalise and give more details in the following definition.

Definition 11 (OBDM framework) *An* OBDM (Ontology-Based Data Management) framework *is characterised by a triple* $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$ *parameterised by the semantics specification* $\mathbb{S}$ *and language specification* $\mathbb{L}$ *such that:*

- $\mathcal{S}$ *is the source schema, possibly encoding integrity constraints,*
- $\mathcal{M}$ *is the set of mappings, i.e., rules that associate database schema with ontology schema of the form defined in Def. 9,*
- $\mathcal{T}$ *is the TBox (ontology),*
- $\mathbb{S} = \langle \mathbb{Q}, \mathbb{U} \rangle$ *is the semantics:* $\mathbb{Q}$ *typically standing for the certain answer semantics, and* $\mathbb{U}$ *is the semantics of ontology updates,*
- $\mathbb{L} = \langle \mathcal{Q}, \mathcal{U} \rangle$ *is the pair of query and update languages over* $\mathcal{T}$ *used to access and manipulate the data.*

After the definition of the OBDM framework, we provide the concrete instantiation by using the database instance $\mathcal{D}$, hence the definition of OBDM system.

Definition 12 (OBDM system) *An* OBDM system *is a pair of* $\langle \mathcal{O}_m, \mathcal{D} \rangle$ *where:*

- $\mathcal{O}_m = \langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$ *is an* OBDM framework*,*
- $\mathcal{D}$ *satisfies integrity constraints, if they are present in* $\mathcal{S}$*.*

Finally, with all the previous definitions in place, we can give the definition of the semantics of OBDM.

Definition 13 (Semantics of OBDM) *An interpretation* $\mathcal{I}$ *is a model of an OBDM system* $\langle \mathcal{O}_m, \mathcal{D} \rangle$ *if*

- $\mathcal{I}$ *is a model of* $\mathcal{T}$*,*
- $\mathcal{I}$ *satisfies* $\mathcal{M}$ *w.r.t.* $\mathcal{D}$*, i.e., satisfies every assertion in* $\mathcal{M}$ *w.r.t.* $\mathcal{D}$*.*

Intuitively, it is clear that an interpretation not only should satisfy TBox axioms, but also the virtual view, which is in turn created by $\mathcal{D}$ and $\mathcal{M}$.

Next, we extend the definition of queries to the OBDM setting.

Definition 14 (Queries over OBDM) *The certain answers to $q(\vec{x})$ over $\langle \mathcal{O}_m, \mathcal{D} \rangle$, denoted with $cert(q, \mathcal{O}_m, \mathcal{D})$, are the tuples $\vec{t}$ of ground object terms (including Skolem functions) and constants from $\mathcal{D}$ such that $\vec{t} \in q^{\mathcal{I}}$, for every model $\mathcal{I}$ of $\langle \mathcal{O}_m, \mathcal{D} \rangle$.*

In order to compute the certain answers in the context of mappings, OBDM systems typically after the rewriting of the query, translate it to SQL query via a translation procedure called *unfolding*.

Example 10 Given the CQ query from Ex. 4, according to the defined mappings in Ex. 9, can be translated to the following SQL query (simplified here):

```
SELECT EmpID
FROM Employee Emp, Department Dept, EmpDept ED
WHERE Emp.ID = ED.EmpID
AND Dept.ID = ED.DEPTID
AND Dept.ID = 102
```

One can observe that the returned tuples as the answer of the query coincide with Ex. 5, i.e., *ans*={joe, anna}. ■

2.2 Semantic Web

The Semantic Web is based on a set of standards: RDF [HPS14] as a graph data model, RDFS [BGe04] and OWL [MGH+12] as ontology languages, and SPARQL [HS13] as a language for querying and manipulating graph data. Since in this dissertation, ideas are drawn from OBDM and *DL-Lite*, these are introduced in a way that is compatible with the notions from the previous section.

2.2.1 RDF

RDF (Resource Description Framework) is a framework for representing data resources on the Web. With resource we mean anything in the world including physical things, documents, abstract concepts etc. RDF consists for three types of terms:

- Resource identifiers (IRIs) I,
- literals L to denote datatype values,
- blank nodes B to denote existent, but unnamed resources; one can think of blank nodes as existentially quantified variables for making statements about existent but unknown resources.

Let us assume there are infinite sets of pairwise disjoint IRIs of I, L and B.

Definition 15 (RDF triple, graph) *A triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an* RDF triple*, where s is the subject, p is the predicate (property) and o is the object value. A finite set of RDF triples is called an* RDF graph*.*

	TBox	RDFS
1	$A' \sqsubseteq A$	A' sc A.
2	$P' \sqsubseteq P$	P' sp P.

	TBox	RDFS
3	$\exists P \sqsubseteq A$	P dom A.
4	$\exists P^- \sqsubseteq A$	P rng A.

	ABox	RDFS
5	$A(x)$	x a A.
6	$P(x, y)$	x P y.

Table 22: *DL-Lite*$_{\text{RDFS}}$ assertions vs. RDF(S), where A, A' denote concept (or, class) names, P, P' denote role (or, property) names, Γ is a set of constants, and $x, y \in \Gamma$. For RDF(S) vocabulary, we make use of similar abbreviations (sc, sp, dom, rng, a) introduced in [MPG07].

2.2.2 Ontology Languages RDFS and OWL

We initially use a minimal definition of RDFS, in line with Muñoz et al. [MPG07], which represents a minimal subset of DL axioms, in order to stay in the intersection of RDFS and DLs.

Definition 16 (RDFS ontology, ABox, TBox, triple store) *We call a set $\mathcal{T}$ of inclusion assertions of the forms 1–4 in Table 22 an* RDFS ontology*, or* (RDFS) TBox*, a set $\mathcal{A}$ of assertions of the forms 5–6 in Table 22 is called an* (RDF) ABox*. Finally, we call a container, or a knowledge base $G = \mathcal{T}_G \cup \mathcal{A}_G$ holding a TBox $\mathcal{T}_G$ and an ABox $\mathcal{A}_G$ an* (RDFS) triple store*.*

In the context of RDF(S), Γ is the set of constants (cf. Table 22), which coincides with the set I of IRIs. We assume the IRIs used for concepts, roles, and individuals to be disjoint from IRIs of the RDFS and OWL vocabularies[9], listed in Table 23, which are used in this dissertation.

In the following, we view RDF and DL notation interchangeably, i.e., we treat any RDF graph consisting of triples without non-standard RDFS vocabulary as a set of TBox and ABox assertions.

Example 11 As a running example, we assume a triple store G with RDF (ABox) data and an RDFS ontology (TBox) (in Turtle syntax [BBLPC13]) as follows:

ABox:

```
:john :worksFor :marketing. :joe :worksFor :finance.
:anna :worksFor :marketing; :worksFor :finance .
```

TBox:

[9] That is, we assume no "non-standard use" [PHDU13] of these vocabularies. While we could assume concept names, role names, and individual constants to be mutually disjoint, we rather distinguish implicitly between them "per use" (in the sense of "punning" [Mot07]) based on their position in atoms or RDF triples.

	Axiom type	RDFS/OWL	Abbreviation
1	type	rdf:type	a
2	subclass	rdfs:subClassOf	sc
3	subproperty	rdfs:subPropertyOf	sp
4	domain	rdfs:domain	dom
5	range	rdfs:range	rng
6	differentFrom	owl:differentFrom	df
7	inverse property	owl:inverse	inv
8	class disjointness	owl:disjointWith	dw
9	property disjointness	owl:propertyDisjointWith	pdw
10	functional property	owl:FunctionalProperty	func

Table 23: RDFS and OWL terms together with their respective IRIs and abbreviations, which are used in this dissertation within axioms.

```
:Employee sc :Person.   :Department sc :Organisation.
:worksFor sp :belongsTo. :worksFor rng :Department; dom :Employee.
```

■

In order to define the semantics of RDFS—as we have previously defined the semantics of $\mathcal{ALCI}$—we rely on the standard notions of (first-order logic) *interpretation*, satisfaction of assertions, and *model* (cf. Def. 5).

Besides RDFS, there exist three ontology profiles (fragments, sublanguages) of *OWL 2*[10] that are more expressive, namely: *OWL 2 EL*[11], *OWL 2 QL*[12] and *OWL 2 RL*[13]. The purpose of these ontology profiles is to have languages that have a good expressivity while at the same time being tractable; the latter does not hold for OWL 2 in general. These profiles are distinguished by their expressivity. They have been tailored to specific use cases, taking into account the complexity of reasoning tasks[14].

OWL 2 EL, QL and RL subsume the constructs of minimal RDFS that we covered: subclass, subproperty, domain and range; and on top of that they also contain, among others, "equivalentClass" ($A_1 \equiv A_2$) and "equivalentProperty" ($P_1 \equiv P_2$) axioms. Note that these two axioms can always be split up in two axioms of subclass ($A_1 \sqsubseteq A_2$, $A_2 \sqsubseteq A_1$) and subproperty ($P_1 \sqsubseteq P_2$, $P_2 \sqsubseteq P_1$) respectively. Now, let us briefly discuss each profile:

[10] https://www.w3.org/TR/owl2-overview/
[11] https://www.w3.org/TR/owl2-profiles/#OWL_2_EL
[12] https://www.w3.org/TR/owl2-profiles/#OWL_2_QL
[13] https://www.w3.org/TR/owl2-profiles/#OWL_2_RL
[14] https://www.w3.org/TR/owl2-profiles/#Computational_Properties

- *OWL 2 EL* is designed for ontologies that consist of big class and/or property hierarchies, using fairly few OWL features, such as transitive properties. A typical use case is the medical ontology SNOMED CT, which has more than hundred thousand classes and properties. Other ontologies in life sciences are also a candidate for this profile. OWL 2 EL is based on the EL++ Description Logic [BBL05].

- *OWL 2 QL* is based on DL-Lite [CDGL$^+$07] while keeping the constructs which make sense in the realm of Semantic Web such as `owl:sameAs`, and at the same time dropping the Unique Name Assumption[15][16] in DL-Lite. As it is based on DL-Lite, it is designed for data-driven applications, i.e., it is used on ontologies where ABox data is large and stored in a database layer managed by DBMS. Notably, this profile contains symmetric and inverse properties.

- *OWL 2 RL* is designed to represent rules in ontologies, which can be leveraged by reasoners and other rule-based reasoning engines. It is used by applications that require scalable reasoning without sacrificing too much expressive power, while being more expressive than RDFS. It is based on Description Logic Programs (DLP) [GHVD03]. Notably, this profile also contains constructs such as inverse and symmetric properties.

For a more thorough treatment of OWL 2 profiles, as well as the restrictions where constructs can be applied, we refer to [Kr12]. Next, we focus our discussion on the SPARQL query language, which can be used to query and update both RDF data and ontologies.

2.2.3 SPARQL Query Language

SPARQL is the standard query language for RDF and is designed to have a similar "look and feel" like SQL, by adopting a similar syntax. SPARQL queries contain a set of triple patterns called *basic graph patterns* (BGPs), which correspond to sets of RDF triples that may contain variables in subject, predicate or object position.

Let $\mathcal{V}$ be a countably infinite set of variables, but now written as '?'-prefixed alphanumeric strings.

Definition 17 (BGP, CQ) *A* basic graph pattern (BGP) *P is a set of atoms of the forms 5–6 from Table 22, where $x, y \in \Gamma \cup \mathcal{V}$. As for UCQs, we denote with $\mathcal{V}(q)$ (or $\mathcal{V}(Q)$) the set of variables from $\mathcal{V}$ occurring in q (resp., Q).*

[15] Unique Name Assumption means that different names (symbols) refer to different entities (objects) in the world [RN10].

[16] Notice that technically UNA is not adopted in OWL 2 QL, but in practice w.r.t. query answering this does not change anything (because there are no equalities between terms that can be inferred in OWL 2 QL).

Example 12 The following SPARQL query is equivalent to the CQ in Ex. 4, under the assumption that `:finance a :Department`:

```
SELECT ?X WHERE { ?X a :Person . ?X :worksFor :finance .}
```

Notice that SPARQL basic graph patterns (BGPs) correspond to CQs in which all variables are *distinguished* (i.e., are answer variables). From the SPARQL perspective, we allow only for BGPs that correspond to standard CQs as formulated over a DL ontology; that is, we rule out BGPs with variables in predicate positions, and on the other hand "terminological" queries, e.g., $\{?x \ \mathsf{sc} \ ?y.\}$. We will relax this latter restriction when we talk about TBox updates (see Sec. 4.5). Also, we do not consider here blank nodes separately[17]. By these restrictions, we can treat query answering and BGP matching in SPARQL analogously and define it in terms of interpretations and models (as usual in DLs). We mention though that there is a fundamental distinction between the two, as SPARQL is based upon the bag/multiset semantics, whereas UCQs are based upon the set semantics.

More complex graph patterns can be formed by combining BGPs with different operators such as OPTIONAL, UNION, FILTER, NOT EXISTS, MINUS and concatenation via a . (point) symbol. Brackets {} are used for grouping patterns.

Following [PAG09], the syntax of SPARQL graph patterns is presented by an algebraic formalism, using operators AND for ., UNION for UNION, OPT for OPTIONAL, MINUS for MINUS, FILTER for FILTER, and SELECT for SELECT.

Definition 18 *A general SPARQL* graph pattern *is defined recursively as follows:*

1. *{} is a graph pattern;*
2. *a tuple from* $(I \cup \mathcal{V}) \times (I \cup \mathcal{V}) \times (I \cup L \cup \mathcal{V})$ *is a graph pattern, called a triple pattern;*
3. *if* P_1 *and* P_2 *are graph patterns, then the expressions* $(P_1 \ \mathsf{AND} \ P_2)$, $(P_1 \ \mathsf{OPT} \ P_2)$, $(P_1 \ \mathsf{UNION} \ P_2)$ *and* $(P_1 \ \mathsf{MINUS} \ P_2)$ *are graph patterns;*
4. *if* P *is a graph pattern and* R *is a SPARQL built-in condition, then the expression* $P \ \mathsf{FILTER} \ R$ *is a graph pattern;*

A SPARQL built-in condition is constructed using elements of the set $I \cup \mathcal{V}$ *and constants, logical connectives* $(\neg, \wedge, \vee)$*, inequality symbols* $(<, \leq, \geq, >)$*, the equality symbol* $=$*, and unary predicates such as* bound*,* isBlank*, and* isIRI*.*

[17] Blank nodes in a triple store may be considered as constants and we do not allow blank nodes in queries, which does not affect the expressivity of SPARQL.

Note that BGPs as per Def. 17 are covered as the restriction of allowing only triple patterns and the operator AND in Def. 18.

The evaluation of SPARQL queries is based on matching graph patterns or BGPs against the triple store[18], resulting in a list of variable bindings. In order to better comprehend the evaluation of SPARQL queries, let us give a few definitions on SPARQL semantics. For further details on the SPARQL semantics, the reader should refer to the official specification [HS13], or to the actual paper that influenced the specification [PAG09].

Definition 19 (SPARQL query answers) *The semantics of SPARQL [PAG09] is defined as a function $[\![\cdot]\!]_G$ that, given a triple store G, takes a graph pattern expression and returns a finite set of mappings μ, where μ is a partial function from the set $\mathcal{V}$ to $(I \cup L \cup B)$. For a triple pattern t we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. The domain of μ, denoted by $dom(\mu)$, is the subset of $\mathcal{V}$ where μ is defined. Two mappings μ_1 and μ_2 are compatible, denoted by $\mu_1 \sim \mu_2$, when for all $x \in dom(\mu_1) \cap dom(\mu_2)$, it is the case that $\mu_1(x) = \mu_2(x)$, i.e., when $\mu_1 \cup \mu_2$ is also a mapping. The mapping with empty domain is denoted by $\mu_\emptyset$, and it is compatible with any other mapping.*

The evaluation of a graph pattern P over a triple store G, denoted by $[\![P]\!]_G$ is defined recursively as follows:

1. *If P is $\{\}$ and G is non-empty, then $[\![P]\!]_G = \mu_\emptyset$.*
2. *If P is $\{\}$ and $G = \emptyset$, then $[\![P]\!]_G = \emptyset$.*
3. *If P is a triple pattern t, then $[\![P]\!]_G = \{\mu | dom(\mu) = var(t)$ and $\mu(t) \in G\}$*
4. *If P is (P_1 AND P_2), then $[\![P]\!]_G = \{\mu_1 \cup \mu_2 \mid \mu_1 \in [\![P_1]\!]_G, \mu_2 \in [\![P_2]\!]_G$ and $\mu_1 \sim \mu_2\}$.*
5. *If P is (P_1 OPT P_2), then $[\![P]\!]_G = \{\mu_1 \cup \mu_2 \mid \mu_1 \in [\![P_1]\!]_G, \mu_2 \in [\![P_2]\!]_G$ and $\mu_1 \sim \mu_2\} \cup \{\mu \in [\![P_1]\!]_G \mid$ for every $\mu' \in [\![P_2]\!]_G : \mu \not\sim \mu'\}$.*
6. *If P is (P_1 UNION P_2), then $[\![P]\!]_G = \{\mu \mid \mu \in [\![P_1]\!]_G$ or $\mu \in [\![P_2]\!]_G\}$.*
7. *If P is (P_1 MINUS P_2), then $[\![P]\!]_G = \{\mu \in [\![P_1]\!]_G \mid$ for every $\mu' \in [\![P_2]\!]_G : \mu \not\sim \mu'$ or $dom(\mu) \cap dom(\mu') = \emptyset\}$.*

The semantics of filter expressions goes as follows; for simplicity we restrict ourselves to the equality symbol only. Given a mapping μ and a built-in condition R, we say that μ satisfies R, denoted by $\mu \models R$, if:

1. R is $bound(?X)$ and $?X \in dom(\mu)$;

[18] As we have defined the triple stores as collection of ABox and TBox data, we refer to them instead of merely graphs.

2. R is $?X = c, ?X = dom(\mu)$ and $\mu(?X) = c$;
3. R is $?X = ?Y, ?X \in dom(\mu), ?Y \in dom(\mu)$ and $\mu(?X) = \mu(?Y)$;
4. R is $(\neg R_1), R_1$ is a built-in condition, and is not the case that $\mu \models R_1$;
5. R is $(R_1 \vee R_2), R_1$ and R_2 are built-in conditions, and $\mu \models R_1$ or $\mu \models R_2$;
6. R is $(R_1 \wedge R_2), R_1$ and R_2 are built-in conditions, and $\mu \models R_1$ and $\mu \models R_2$.

Definition 20 (The semantics of FILTER) *Given a triple store G and a filter expression P FILTER R, then:*

$$[\![(P \text{ FILTER } R)]\!]_G = \{\mu \in [\![P]\!]_G \mid \mu \models R\}.$$

Example 13 The following SPARQL query is equivalent to the CQ in Ex. 4, and to SPARQL query in Ex. 12 (now written with FILTER):

```
SELECT ?X WHERE { ?X a :Person . ?X :worksFor ?Y .
                  ?Y a :Department . FILTER (?Y=:finance) }
```

We conclude the discussion on the SPARQL query answers, and now we extend it in the context of entailment regimes.

2.2.4 SPARQL Entailment Regimes

In SPARQL terminology, an *entailment regime* with respect to an ontology language, refers to an extension of the concept of SPARQL query answers, such that query answers are extended in the sense that BGPs, i.e., CQs also return those answers entailed with respect to inferences in the given ontology language. That is, entailment regimes are the equivalent of certain answers in ontologies, which in turn are defined by query rewriting in DL-Lite.

For the case of RDFS, the SPARQL specification defines an entailment regime that takes into account RDFS inferences plus other aspects such as axiomatic triples etc. [GOH+13].

For the purposes of this dissertation, we slightly simplify the entailments under RDFS in the spirit of [MPG07], i.e., by RDFS entailment we mean only certain answers with respect to the minimal set of entailment rules given by Munoz et al., cf. Fig. 23. Under this simplifying view of RDFS entailments, the RDFS SPARQL entailment regime can actually be defined in terms of either query rewriting or rule-based materialisation.

First, let us define query answering under RDFS entailment as query rewriting. In order to do that, we need to borrow the rewriting techniques from DL-Lite and OBDM. Precisely, we will leverage upon *PerfectRef* (Alg. 2.1) which is an algorithm that takes as input a DL-Lite TBox $\mathcal{T}$ and a CQ q and returns an UCQ q'.

The following definition formalises the previous discussion.

Definition 21 (Query Rewriting) *Given a CQ q and a triple store $G = \mathcal{T}_G \cup \mathcal{A}_G$, then $rewrite(q, \mathcal{T}_G)$ is the UCQ resulting from applying PerfectRef Alg. 2.1 to q and G.*

The goal rewrite operator $gr(g, I)$ in *PerfectRef* also in the case of properties introduces fresh variables in a non-relevant position (cf. Table 21). In the ontology language we are currently elaborating, i.e., minimal RDFS, the "Reduce" operator (lines 12-16, Alg. 2.1) which is tailored to DL-Lite, it is not applicable herein (note that in Ex. 6 after the reduce step one can apply mandatory participation axioms, and such constructs are not contained in RDFS ontology language). In the end, we get a rewritten query UCQ obtained from the original CQ that is able to return implicit answers.

Definition 22 (Query answers) *For a CQ q (or, UCQ Q, resp.) and a triple store G, a substitution θ from variables in $\mathcal{V}(q)$ to constants in Γ such that $q'\theta \in rewrite(q, \mathcal{T}_G)\theta$ is true (or, there exists a $q \in Q$ with $q'\theta$ is true) in every model of G is called an* answer *(under RDFS Entailment) to q, and we denote the set of all answers by $ans_{rdfs}(q, G)$ (or simply $ans(q, G)$). The set of answers to a UCQ Q is $\bigcup_{q \in Q} ans(q, G)$.*

Example 14 (cont'd) The rewriting $q' = rewrite(q, \mathcal{T}_G)$ of the query in Ex. 13 according to *PerfectRef* with respect to $\mathcal{T}_G$ as a DL TBox written in SPARQL yields:

```
SELECT ?X
WHERE {
      { ?X a :Person . ?X :worksFor ?Y . ?Y a :Department .
                                              FILTER (?Y=:finance) }
      UNION
      { ?X a :Employee . ?X :worksFor ?Y . ?Y a :Department .
                                              FILTER (?Y=:finance) }
      UNION
      { ?X :worksFor ?Y1 . ?X :worksFor ?Y. ?Y a :Department .
                                              FILTER (?Y=:finance) }
      UNION
      { ?X a :Person . ?X :worksFor ?Y . ?X1 :worksFor ?Y .
                                              FILTER (?Y=:finance) }
      UNION
      { ?X a :Employee . ?X :worksFor ?Y . ?X1 :worksFor ?Y .
                                              FILTER (?Y=:finance) }
      UNION
      { ?X :worksFor ?Y1 . ?X :worksFor ?Y. ?X1 :worksFor ?Y .
                                              FILTER (?Y=:finance) }
}
```

Indeed, this query evaluated over the ABox returns both `:joe` and `:anna`. This rewriting returns certain answers even in the case where mappings as in Ex. 9 are present [CDGL$^+$07], modulo a further unfolding step. ■

$$\frac{?C \text{ sc } ?D. \quad ?S \text{ a } ?C.}{?S \text{ a } ?D.} \qquad \frac{?P \text{ dom } ?C. \quad ?S\ ?P\ ?O.}{?S \text{ a } ?C.} \qquad \frac{?C \text{ sc } ?D. \quad ?D \text{ sc } ?E.}{?C \text{ sc } ?E.}$$

$$\frac{?P \text{ sp } ?Q. \quad ?S\ ?P\ ?O.}{?S\ ?Q\ ?O.} \qquad \frac{?P \text{ rng } ?C. \quad ?S\ ?P\ ?O.}{?O \text{ a } ?C.} \qquad \frac{?P \text{ sp } ?Q. \quad ?Q \text{ sp } ?R.}{?P \text{ sp } ?R.}$$

Figure 23: Minimal RDFS rules from [MPG07]

An alternative[19] to rewriting is to materialise all inferences in the triple store, such that the original query can be used 'as is', for instance using the minimalistic inference rules for RDFS from [MPG07][20] shown in Fig. 23.

The materialisation of a triple store, denoted $mat(G)$, is computed by performing a deductive closure for all the triples subject to a given ontology language. In this case, we provide a set of minimal inference rules in RDFS (cf. Fig. 23)—which can be represented as Datalog rules (cf. Fig. 24)—that are used to compute the materialisation of a triple store. These rules are fired recursively until no new triple can be entailed by the given set of explicit triples plus the newly derived implicit triples. We will provide a more formal definition of $mat(G)$ in Chapter 4.

Example 15 (Materialised G) The materialised version of G would contain the following triples – for conciseness only assertional implied triples are shown here, that is triples from the four leftmost rules in Fig. 23.

```
:john a :Employee ; :Person .
:joe a :Employee ; :Person .
:anna a :Employee ; :Person .
:finance a :Department ; :Organization .
```

```
triple(S, a, D) :- triple(S, rdfs:subClassOf, D), triple(S, a, C) .
triple(S, Q, O) :- triple(P, rdfs:subPropertyOf, Q), triple(S, P, O) .
triple(S, a, C) :- triple(P, rdfs:domain, C), triple(S, P, O) .
triple(O, a, C) :- triple(P, rdfs:range, Q), triple(S, P, O) .
triple(C, rdfs:subClassOf, E) :- triple(C, rdfs:subClassOf, D),
                                  triple(D, rdfs:subClassOf, E) .
triple(P, rdfs:subPropertyOf, R) :- triple(P, rdfs:subPropertyOf, Q),
                                    triple(Q, rdfs:subPropertyOf, R) .
```

Figure 24: In order to represent the minimal RDFS rules Fig. 23 in Datalog, we need an auxiliary predicate called "triple".

[19] This alternative is viable for RDFS, but not necessarily for more expressive DLs.

[20] These rules correspond to rules 2), 3), 4) of [MPG07]; they suffice since we ignore blank nodes.

```
:marketing a :Department ; :Organization .
:john :belongsTo :marketing . :joe :belongsTo :finance .
:anna :belongsTo :marketing . :anna :belongsTo :finance .
```

On the materialised triple store, the query from Ex. 13 would return the expected results (without rewriting). ■

The next result follows immediately from, e.g., [MPG07, GHV11, CDGL$^+$07] and shows that query answering under RDFS can be done by either query rewriting or materialisation.

Proposition 2 *Let $G = \mathcal{T}_G \cup \mathcal{A}_G$ be a triple store, q a CQ, and $\mathcal{A}'_G$ the set of ABox assertions in $mat(G)$. Then, $ans(q, G) = ans(rewrite(q, \mathcal{T}_G), \mathcal{A}_G) = ans(q, \mathcal{A}'_G)$.*

Various triple stores (e.g., BigOWLIM [BKO$^+$11]) perform ABox materialisation directly upon loading data. However, such triple stores do not necessarily materialise the TBox: in order to correctly answer UCQs as defined above, a triple store actually does not need to consider the two rightmost rules in Fig. 23. Accordingly, we will call a triple store, or an ABox *materialised* if in each state it is always materialised.

On the other extreme, we find triple stores that do not store *any* redundant ABox triples. Such triple stores are space-efficient to the expense of query rewriting and evaluation, given that they have to rewrite a query and evaluate it in order to obtain the complete set of results including the implicit triples.

By $red(G)$ we denote the hypothetical operator that produces the reduced "core" of a triple store G. Note that computing the core is not always feasible in a more expressive ontology languages, but we show that it is possible in the case of the minimal RDFS ontology language and we give a possible implementation in Sec. 7.1. Similarly, we will call a triple store, or an ABox *reduced* if in each state it is always reduced. We will provide a more formal definition of $red(G)$ in Chapter 4.

An example of reduced store is given in the following.

Example 16 (Reduced G) The reduced version of G from Ex. 15 contains the following triples:

```
:john :worksFor :marketing. :joe :worksFor :finance.
:anna :worksFor :marketing; :worksFor :finance .
```

■

Note that in this example, the reduced G contains only the triples that are not redundant, i.e., hence by removing all the implicit triples using the operator $red(G)$.

2.2.5 SPARQL Update

Finally, we introduce the notion of a SPARQL update operation.

Definition 23 (SPARQL update operation) *Let P_d and P_i be BGPs, and P_w a BGP or UNION pattern. Then an* update operation *$u(P_d, P_i, P_w)$ has the form*

$$\textbf{\textit{DELETE}}\ \ P_d\ \ \textbf{\textit{INSERT}}\ \ P_i\ \ \textbf{\textit{WHERE}}\ \ P_w.$$

Intuitively, the semantics of executing $u(P_d, P_i, P_w)$ on G, denoted as $G_{u(P_d,P_i,P_w)}$[21] is defined by interpreting both P_d and P_i as "templates" to be instantiated with the solutions of $ans(P_w, G)$, resulting in sets of ABox statements $\mathcal{A}_d$ to be deleted from G, and $\mathcal{A}_i$ to be inserted into G. A naïve update semantics follows straightforwardly.

Definition 24 (Naïve update semantics) *Let $G = \mathcal{T}_G \cup \mathcal{A}_G$ be a triple store, and $u(P_d, P_i, P_w)$ an update operation. Then, the* naïve update of G with $u(P_d, P_i, P_w)$, *denoted $G_{u(P_d,P_i,P_w)}$, is defined as $(G \setminus \mathcal{A}_d) \cup \mathcal{A}_i$, where $\mathcal{A}_d = \bigcup_{\theta \in ans(P_w,G)} ground(P_d\theta)$, $\mathcal{A}_i = \bigcup_{\theta \in ans(P_w,G)} ground(P_i\theta)$, and $ground(P)$ denotes the set of ground triples in pattern P.*

We will often refer to naive update semantics using the notation G_u^{naive}, whereas for any other semantics *sem*, we will similarly use the notation G_u^{sem}.

2.2.6 Mapping Relational Databases to RDF

In order to get data out of relational databases, in general one has to rely on SQL queries involving complex constructs and join operators, in addition to simply mapping each column of a table to a symbol from vocabulary. OBDM systems allow to define mappings to relational databases either in their native mapping language, or in a more well-established mapping language endorsed by W3C such as direct mapping[22] and R2RML[23].

Example 17 Table 24 shows the mappings from Ex. 9 in the native Ontop syntax [CCK+17], where results from a query are mapped to a set of RDF triples having variables in either subject or object positions.

[21] Note that in the following, we write just G_u for $G_{u(P_d,P_i,P_w)}$ as a a shortcut or when clear from the context.

[22] `https://www.w3.org/TR/rdb-direct-mapping/`

[23] `https://www.w3.org/TR/r2rml/`

m_1 :	:{EMPID} :worksFor :{DEPTID}	←	**SELECT** EMPID, DEPTID **FROM** empdept
m_2 :	:{ID} a :Employee . :{ID} :name :{EmpName}	←	**SELECT** ID, EmpName **FROM** employee emp, empdept ed **WHERE** emp.ID=empdept.EMPID **AND** emp.Salary < 2500
m_3 :	:{ID} a :Manager . :{ID} :name :{EmpName}	←	**SELECT** ID, EmpName **FROM** employee emp, empdept ed **WHERE** emp.ID=empdept.EMPID **AND** emp.Salary >= 2500
m_4 :	:{ID} a :Department . :{ID} :deptName :{DeptName}	←	**SELECT** ID, DeptName **FROM** department dept, empdept ed **WHERE** dept.ID=ed.DEPTID

Table 24: Mappings in Ontop syntax

When we will be discussing the other related approaches in Chapter 8, we will refer to other examples of OBDM with mappings expressed in R2RML (cf. Fig. 84), or in another well-known mapping language called D2RQ[24] (cf. Fig. 82).

Allowing SQL queries in the rule bodies in the mapping language enables the flexibility of having very complex view definitions. While this makes querying and data retrieval more convenient, on the other hand it makes view updates challenging. As known from database theory, in the cases where as a view definition we have a SQL query using joins of two different tables, one can easily run into the "view update problem." In other words, this means that the translated update would not precisely reflect the intended update on the view (for motivational examples see Sec. 3.1). For that reason, to avoid such cases, it is common practice to either restrict oneself to one-to-one mappings or to no mappings at all. The latter case boils down to the case of triples stored in a triple store (cf. Fig. 25).

Proposition 3 (Triple store as OBDM) *Given a triple store $G = \mathcal{T}_G \cup \mathcal{A}_G$, then G can be considered as an OBDM, with a singular underlying (ternary) relational table* t *(using the subject-predicate-object format* [s, p, o]*), where there is a mapping of the form:*

```
triple(S, P, O) :- t(S, P, O)
```

In fact, nowadays we have triple stores, e.g., Stardog, that can play the role of a "genuine" OBDM, i.e., they access heterogeneous data stored in the relational layer or MongoDB, without explicitly translating them to RDF, and on top of that they provide query answering that is based on query rewriting. Moreover, such triple stores can mix virtual and materialised graphs, in this way creating a hybrid approach to data management.

Fig. 25 depicts an OBDM where mappings are one-to-one. The mappings are displayed in the same way as in [HRG10], where boxes indicate different mappings, i.e., whether

[24] http://d2rq.org

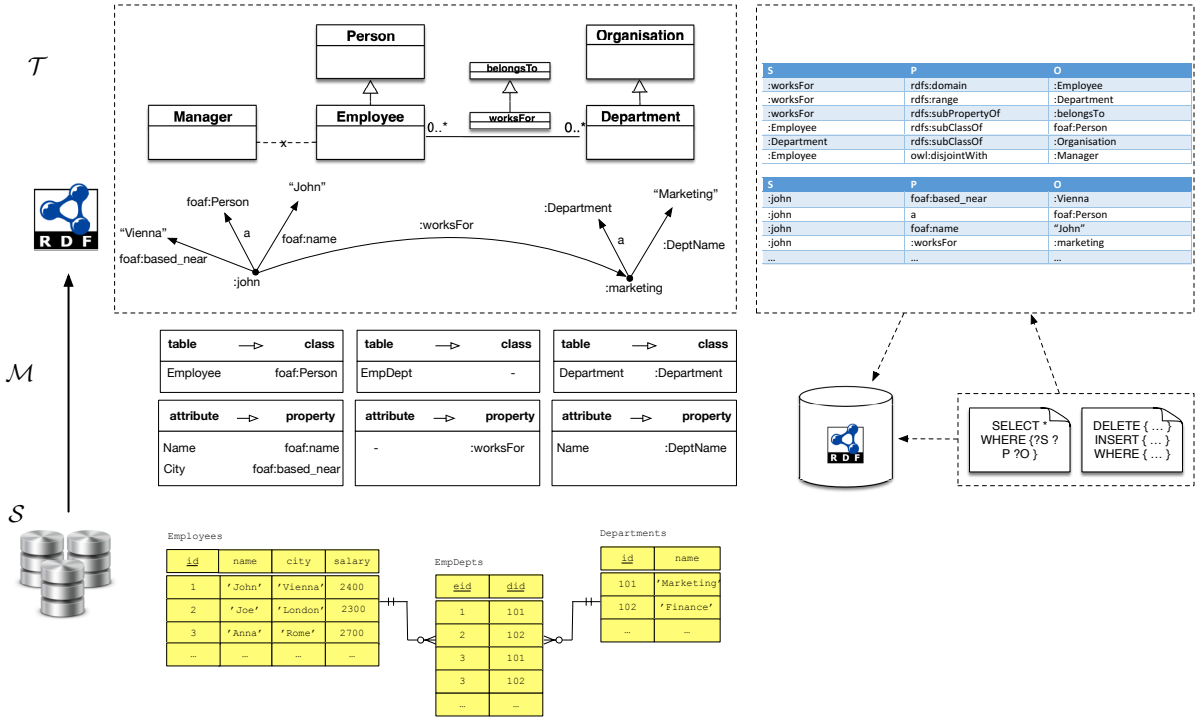


Figure 25: The figure on the left side visually depicts all the components of OBDM, yielding a view with ABox (or a graph) and TBox. On the right side triples are displayed serialised in Turtle that correspond to the left side. These triples can either be materialised in a dedicated triple store in ETL fashion, or they can be used simply as a view. SPARQL is used to query and update the triples.

the mapping is from table to a class, or from attribute to a property respectively (note that not all generated triples are displayed in the figure, but those can be easily deduced by the reader from the database instance and the definition of mappings).

Direct mappings such as *direct mapping* endorsed by W3C, or $\mathcal{DM}$ [Seq16], also fall in the category of one-to-one mappings, given that RDF triples are generated with a predefined vocabulary in an unambiguous way from the database schema.

2.3 Formula-Based Approaches to Updates

When updating an ontology a main difficulty lies in keeping the updated ontology consistent. Several approaches to preserve the update intuition while keeping consistency have been proposed in the literature.

In this dissertation, based on [Win05], we distinguish between two means of performing an update, which are orthogonal to each other: (1) the users decide about the semantics of the update, i.e., what to delete and insert from the knowledge base – the easier

option, and (2) performing an update intentionally by a formula which satisfies the new state of the world, and letting algorithms accomplish that change in the knowledge base. The latter approaches as of [Win05] are accordingly divided into *model-based* and *formula-based* and they both differently affect the outcome of the knowledge base update. Given that model-based approaches to updates in general do not give intuitive results [CKNZ10], we elaborate only formula-based approaches herein.

In formula-based approaches to updates, the objects of change are sets of formulas. The challenge in evolution in formula-based approaches of ontology $\mathcal{O}$ with $\mathcal{U}$, denoted by $\mathcal{O} \diamond \mathcal{U}$, is to find a unique maximal subset $\mathcal{O}'$ of $\mathcal{O}$, which is consistent with the new knowledge $\mathcal{U}$. In the literature [Win05], there are two main approaches to formula-based semantics when dealing with new information to be incorporated into the existing ontology: *Cross-Product* and *WIDTIO* (When In Doubt Throw It Out).

Definition 25 (Cross-Product) *(Adapted from [CKNZ10]) Let $\mathcal{O}$ be an ontology in the materialised state (i.e., deductively closed), and let $\mathcal{U}$ be the new information to be incorporated. Furthermore, let $\mathcal{M}(\mathcal{O},\mathcal{U})$ be the set of maximal subsets of $\mathcal{O}$, i.e., $\mathcal{O}' \subseteq \mathcal{O}$, that are consistent with $\mathcal{U}$. Then, the* Cross-Product *(abbr. CP) is the disjunction of these maximal subsets, defined as follows:*

$$\mathcal{O}_{CP} = \bigvee_{\mathcal{O}' \in \mathcal{M}(\mathcal{O},\mathcal{U})} \left(\bigwedge_{\phi \in \mathcal{O}'} \phi \right)$$

The disadvantage of *CP* is that in some logics (e.g., DL-Lite) the result can not be captured due to the disjunction operator not being expressible within DL-Lite, plus the result might be of exponential size in the worst-case.

Definition 26 (WIDTIO) *(Adapted from [CKNZ10]) Let $\mathcal{O}$ be the ontology in the materialised state, and let $\mathcal{U}$ be the new information to be incorporated. As before, let $\mathcal{M}(\mathcal{O},\mathcal{U})$ be the set of maximal subsets of $\mathcal{O}$, i.e., $\mathcal{O}' \subseteq \mathcal{O}$, which are consistent with $\mathcal{U}$. Then, the* WIDTIO (When In Doubt Throw It Out) *approach is to take the intersection of all the maximal subsets, defined as follows:*

$$\mathcal{O}_{WIDTIO} = \bigcap_{\mathcal{O}' \in \mathcal{M}(\mathcal{O},\mathcal{U})} \mathcal{O}'$$

Although using *WIDTIO* there are no issues regarding capturing the result of the update as it was the case with *CP*, the disadvantage is that it might throw away too many axioms.

Example 18 (Cross-Product and WIDTIO) Given the following ontology $\mathcal{O}$ consisting of an empty ABox and TBox $\mathcal{T} = \{Manager \sqsubseteq Employee, TopManager \sqsubseteq Manager\}$, and an update $\mathcal{U} = \{TopManager \sqsubseteq \neg Employee\}$.

The update $\mathcal{U}$ posed over the ontology $\mathcal{O}$ makes the concept $TopManager$ unsatisfiable, while the ontology is still consistent though, i.e., has at least one model. If we want to apply a stronger notion of consistency where each concept has at least one individual assigned after an update (see *coherence* [CKNZ10]), then one can either drop $Manager \sqsubseteq Employee$, or $TopManager \sqsubseteq Manager$. Thus, $\mathcal{M}(\mathcal{O},\mathcal{U}) = \{\mathcal{O}'_{(1)}, \mathcal{O}'_{(2)}\}$, where $\mathcal{O}'_{(1)} = \mathcal{O} \setminus \{Manager \sqsubseteq Employee\}$ and $\mathcal{O}'_{(2)} = \mathcal{O} \setminus \{TopManager \sqsubseteq Manager\}$.

Then, according to *Cross-Product* and *WIDTIO*, the results respectively are:

$$\begin{aligned}\mathcal{U} \cup \mathcal{O}_{CP} &= \mathcal{U} \cup (\mathcal{O} \setminus \{TopManager \sqsubseteq Manager\}) \vee (\mathcal{O} \setminus \{Manager \sqsubseteq Employee\}) \\ \mathcal{U} \cup \mathcal{O}_{WIDTIO} &= \mathcal{U} \ \cup \ (\mathcal{O}'_{(1)} \ \cap \ \mathcal{O}'_{(2)}) \\ &= \mathcal{U} \ \cup \ \mathcal{O} \ \setminus \ \{TopManager \sqsubseteq Manager, Manager \sqsubseteq Employee\}\end{aligned}$$

■

As two further algorithms from the literature for ontology and view updates, we introduce *DRed* and *Counting*, which also fall into the category of formula-based approaches.

The seminal paper [GMS93] introduces two algorithms *DRed* (short for Delete and Rederive) and *Counting* for maintaining materialised views where updates occur in the underlying relational sources. *DRed* can also be used in top-down approach of propagating updates, i.e., in the same fashion as updates over views.

DRed is an algorithm that maintains the materialisation of recursive views (defined in terms of a Datalog program) with negation and aggregation. The *DRed* algorithm essentially computes:

1. Delete a superset of the derived (inferred) tuples using semi-naive evaluation[25], so-called "overestimation",
2. re-insert deleted tuples that have an alternative derivation, i.e., "re-derive",
3. insert new tuples plus the corresponding derived tuples (using semi-naive evaluation).

Step 3 is used only in the case of updates involving both deletes and inserts.

We illustrate the approach by a simple example when we consider the ontology from Ex. 3 as the "program" defining a recursive view.

Example 19 (Incremental Update using *DRed*) Given $G = \mathcal{T}_G \cup \mathcal{A}_G$ as in Ex. 3 in an already materialised state. Let us consider the following update u:

[25] Semi-naive evaluation keeps track of the derived atoms, such that there is no need to do the forward-chaining from the scratch [Ull88].

```
DELETE { :john a :Employee .}
INSERT { :john a :Manager . }
```

In Step 1, *DRed* deletes:
```
DELETE { :john a :Employee . :john a :Person . }
```

In Step 2, *DRed* due to application of `:john :worksFor :marketing`, re-derives:
```
INSERT { :john a :Employee . :john a :Person . }
```

In Step 3, *DRed* inserts:
```
INSERT { :john a :Manager . :john a :Person .}
```

■

Note that the output of the algorithm renders the ontology inconsistent, as *DRed* unlike *Cross-Product* or *WIDTIO*, is not able to deal with inconsistencies in general.

For non-recursive views, *DRed* can be used as well, but *Counting* is more efficient in that respect. *Counting* stores the count of derived tuples in the materialised view, and thus a tuple would be removed from the view only if its respective count value is equal to 0, meaning that there exists no tuple supporting it. The drawback of *DRed* is the delete step which computes an overestimation of the triples to be deleted, which is fixed afterwards by the rederive step. To circumvent this drawback, the *Counting* algorithm takes a different approach by storing along each triple also the number of direct derivations. Thus, in the delete step, deleting a triple would be done by first decrementing the count value of the triple by 1. Also, all the directly dependent triples entailed by rules—as in the case of *DRed*—would be decremented by 1. The operation goes on for n iterations, because the deleted triples (triples with count=0 in the current step) could also decrement other triples that are entailed using the rules, and the process goes on, in the end resulting with deleted triples if the respective count values are 0. The *Counting* algorithm does not need the "rederive" step as *DRed*, to the expense of keeping track the counts. Hence, it distinguishes which triples are to be deleted (and thus are not derivable from $mat(G) \setminus \mathcal{A}_d$) and the triples that persist (that are derivable, i.e., having alternative derivation).

Example 20 (*Counting* algorithm) Given $G = \mathcal{T}_G \cup \mathcal{A}_G$ as in Ex. 3 in a materialised state $G = mat(G)$. Fig. 26 depicts the counts denoted in circle for the respective assertions, i.e., triples. TBox triples always have count=1, whereas ABox triples depend on the number of times they are derived based on the respective rules. Now, deleting u=**DELETE** { `:anna :worksFor :finance` } would delete all the derived (ABox) triples connected via an arrow having count=1 – as their respective count would then be decreased to 0. However, the triple `:anna a :Person` would persist as its count value would then be 1. On the other hand, deleting u=**DELETE** {`:john :worksFor :marketing`} would indeed delete also the triple `:john a :Person`. ■

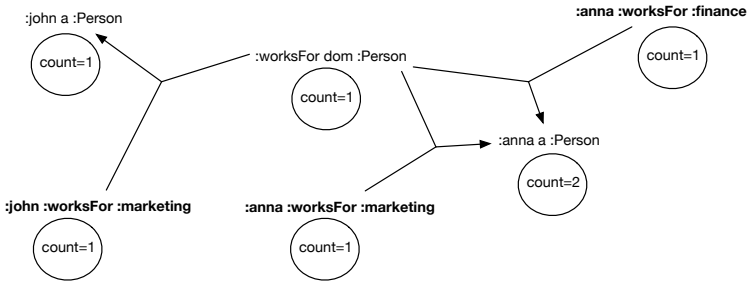


Figure 26: The *Counting* algorithm keeps track of the count of directly derived triples. Explicit ABox triples are denoted in bold, whereas implicit ABox triples here are derived by the rules in the Fig. 23. Note here that the triple `:anna a :Person` is derived twice (hence its count=2), derived from the triples `:anna :worksFor :marketing` and `:worksFor dom :Person`, as well as `:anna :worksFor :finance` and `:worksFor dom :Person` respectively.

CHAPTER 3

OBDM Updates: Background and Desiderata

Suppose that we would like to maintain a set of facts and axioms in a knowledge base, in order to capture the state of the world. As we observe the world and as the world changes, we need a means to update the set of facts and axioms in the knowledge base in order to adjust to the state of the world. A problem arises if we have a knowledge base and we have to insert, for instance `:john a :Manager`, and then in order to perform the update we have to get rid of the old and contradicting facts, such as `:john a :Employee`, given that we have an axiom asserted of type `:Employee owl:disjointWith :Manager` in the knowledge base already.

The problem of updates has been extensively studied in different communities such as databases, AI and philosophy. Even though at the highest level of abstraction they are concerned with similar issues, updating a knowledge base has always been associated with counter-intuitive results, e.g. as in view updating, view maintenance, description logic updates, or belief revision. In each of these fields, which tackle the problem from different perspectives and unique settings, problems occur with "side-effects", complexity of computing the update of a materialised view, inexpressibility and non-deterministic outcome of the knowledge base after an update, non-existence of practical algorithms for belief revision and so on.

In this dissertation we address the problem of updates in the context of Ontology-Based Data Management [Len18]. As described in Chapter 2, OBDM is a framework which consists of *database schemas* $\mathcal{S}$, *mappings* $\mathcal{M}$, a common *ontology* $\mathcal{T}$ and concerns answering queries $\mathcal{Q}$ posed over $\mathcal{T}$. In this framework, besides *queries* $\mathcal{Q}$, *updates* $\mathcal{U}$ are also considered. We denote the OBDM framework using a triple $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$ (cf. Def. 11) with two parameters $\mathbb{S}$ and $\mathbb{L}$ denoting the semantics and languages respectively used for querying and updating the data. Fig. 31 shows all these components of OBDM, where depending

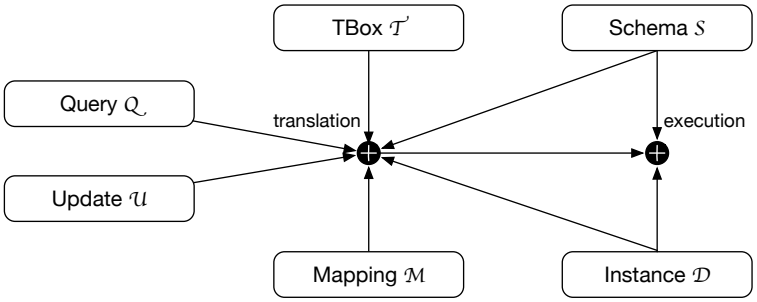


Figure 31: The components of the OBDM framework $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$ with $\mathbb{S} = \langle \mathbb{Q}, \mathbb{U} \rangle$ and $\mathbb{L} = \langle \mathcal{Q}, \mathcal{U} \rangle$, involved in the translation of the queries from $\mathcal{Q}$ and updates from $\mathcal{U}$, and their execution over schema $\mathcal{S}$ and instance $\mathcal{D}$.

on the approach or the semantics, given a query or an update, TBox or/and mappings are taken into account, as well as schema and instances if mappings are applicable, e.g., through the unfolding step. Firstly, the *translation* (aka. *rewriting*) step takes place that could involve all the components. After the translation is computed, the *execution* step takes place over both schema and instances. Regarding the semantics $\mathbb{S}$ for queries, it is common to consider the *certain answer semantics*, although this is not the only viable option [Lib14]. For updates there is no standard semantics as of yet, even in the case of updates over ontologies in absence of mappings [FMK$^+$08, SL12, CKNZ10].

In the following, we motivate the problem of updates in OBDM by illustrating it via a set of examples.

3.1 Update Challenges in OBDM: Motivational Examples

So far, we have seen only the behaviour of queries for well-adopted certain answer semantics. On top of that, the SPARQL 1.1 Entailment Regime specification clearly defines how SPARQL endpoints should treat entailment in the context of queries. In Chapter 2, we have seen how returning entailed query answers can be achieved either using rewriting (feasible in the context of RDFS and DL-Lite), or materialisation in the context of less expressive logics (such as RDFS).

However, regarding updates, there exists no generally accepted standard semantics. First of all, the SPARQL 1.1 Update specification leaves it open how SPARQL endpoints should treat entailment regimes other than simple entailment in the context of updates. As a consequence, different approaches and tools adopt an ad-hoc implementation on treating updates plus entailment.

Example 21 (Implicit update) Given the ABox and TBox data as in the running example Ex. 11, the following SPARQL update operation tries to delete implied triples and to (re-)insert implied triples, based on the instantiations on the WHERE clause that

might also instantiate implied triples:

```
DELETE { ?X a :Employee. }
INSERT { ?Y a :Department. }
WHERE { ?X :belongsTo ?Y. }
```

In the context of instantiations where entailment is taken into account for querying (WHERE clause), then for $\mu(?X)$=`:john`, $\mu(?Y)$=`:marketing`, we get the atomic update:

```
DELETE { :john a :Employee. }
INSERT { :marketing a :Department. }
```

Current OBDM systems have no systematic means for deleting and inserting such implied triples, or for dealing with *implicit updates*. ■

Furthermore, an OBDM system that manages a TBox with a more expressive ontology language beyond RDFS where inconsistencies could occur, first of all, has to take care of such inconsistencies. An OBDM system should for instance, among other desirable properties which we are going to discuss, always guarantee that after an update the resulting state is consistent. Otherwise, querying an inconsistent OBDM yields unexpected and unsafe results.

What is more challenging for languages such as SPARQL/Update is that inconsistencies w.r.t. $\mathcal{T}$ might occur in the instantiations of the update itself alone. In this dissertation we will call these *intrinsic inconsistencies* (see Chapter 5).

Example 22 (Intrinsically inconsistent update) Given an OBDM that maintains the following triples, i.e., facts asserting two people working for each other:

```
:john :worksFor :jane . :jane :worksFor :john .
```

with an ontology $\mathcal{T}$ as follows:

```
:belongsTo rdfs:domain :Employee .
:belongsTo rdfs:range :Manager .
:Employee owl:disjointWith :Manager .
```

Given the following SPARQL update operation:

```
INSERT { ?X :belongsTo ?Y. }
WHERE { ?X :worksFor ?Y. }
```

we get pairs of instantiations $\mu(?X, ?Y) = \{(:\texttt{john}, :\texttt{jane}), (:\texttt{jane}, :\texttt{john})\}$, when instantiating the INSERT template, which in turn now due to domain and range constraints, both `:john` and `:jane` happen to be of type *Manager* and *Employee*, a cue for inconsistency. Note that, as we will see, contrary to Ex. 23, this update clashes w.r.t. $\mathcal{T}$ alone, and not with the old state of the OBDM.

Current OBDM systems have no systematic means of dealing with such *intrinsically-inconsistent updates*. ■

Example 23 (Inconsistent update) Given the ontology and data as in the running example Ex. 11, the following SPARQL update operation tries to insert a triple which is inconsistent w.r.t. $\mathcal{T}$ and the old state of OBDM:

```
INSERT { ?X a :Manager. }
WHERE { ?X :belongsTo :finance. ?X :belongsTo :marketing. }
```

After instantiation $\mu(?X) = :\texttt{anna}$—on an OBDM using query entailment—we have a clash as `:anna` is now both an *Employee* and *Manager*. Thus, OBDM should resolve the inconsistency by either inserting `:anna a :Manager` and deleting the contradicting implied triple `:anna a :Employee` (meaning also deleting all other triples that entail this triple), or alternatively by dropping the update altogether and thus preserving `:anna a :Employee`.

Current OBDM systems have no systematic means of dealing with such *inconsistent updates*. ■

So far, we considered issues arising from the interplay of update operations with the ontology and assertions in OBDM. Now, we consider the issues where mappings are also present, which is typical for OBDM systems:

Example 24 (Ambiguous update) Given an OBDM $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$ system, with the schema $\mathcal{S}$:

$$\textit{employees}[\texttt{EMPNAME}:\textbf{STRING}],$$
$$\textit{customers}[\texttt{CUSTNAME}:\textbf{STRING}],$$

mappings $\mathcal{M}$ (with Skolem functions *emp* and *cus*):

```
Employee(emp(X)) :- employees(X) .
Customer(cus(X)) :- customers(X) .
```

and TBox $\mathcal{T}$:

```
:Employee sc :Person .
:Customer sc :Person .
```

Now, consider the following update:

```
INSERT { :anna a :Person . }
```

In order for this update to be propagated to the relational layer, it should be first qualified as either *Employee* or *Customer*, and one has also to take into account which one of the (ambiguous) mappings to choose. This holds because there are no explicit mappings for class *Person* to the database schema, whereas there are mappings for classes *Employee* and *Customer*, which both are subclasses of *Person*. If we would qualify `:anna` as an *Employee*, then the triple `:anna a :Employee` would be a side-effect, i.e., it would be considered as an unintended fact and not meant by the original update. Still, despite having as a side-effect the triple `:anna a :Employee` or `:anna a :Customer`, we have to opt for one of the options in order to propagate the update.

Current OBDM systems have no systematic means of dealing with such *ambiguous* updates. ■

3.2 Desiderata (Postulates) for Updates in OBDM

As discussed previously, the goal of OBDM systems is to propagate updates u into the instance $\mathcal{D}$, such that the knowledge base derived from $\mathcal{D}$, the mapping $\mathcal{M}$ and the TBox $\mathcal{T}$ reflects u in some reasonable way. Below, we define this intuition as a number of *desiderata* that an OBDM implementation should satisfy.

No matter what semantics you choose for updates, it should adhere to certain desiderata in order for the semantics to be *rational*. For instance, in case of deletions it is not enough for an update semantics to just delete a fact without also deleting all the facts that entail this fact. Alchourrón, Gärdenfors and Makinson (AGM) provided the most widely known theoretical framework [AGM85] consisting of a set of desiderata a.k.a. *postulates* for update operators in the context of belief revision. These postulates provide the essence describing a rational belief revision operator.

In AGM theories are represented by *Belief Sets* B (in some logical language), containing deductively closed sets of sentences ϕ: $B = mat(B) = \{\phi \mid B \vdash \phi\}$. The process of belief revision is a function that maps a theory B and a sentence ϕ to a new theory $B * \phi$.

According to [Pep08], a guiding intuition in formulating those postulates has been the principle of *minimal change* according to which a rational agent changes her belief as little as possible while incorporating the new update in a consistent manner.

According to the AGM framework, there exist three types of theory-change:

- *Expansion* (adding consistent information) $B + \phi$
- *Revision* (adding inconsistent information) $B * \phi$
- *Contraction* (deleting information) $B - \phi$

Expansion is defined straightforwardly:

$$B + \phi = mat(B \cup \{\phi\})$$

On the other hand, the definitions of revision and contraction are not clear, thus different approaches are possible that have to satisfy certain postulates. In the following, AGM postulates are listed that an update operator should satisfy. Postulates 1–6 for both revision and contraction are the *basic postulates*, whereas 7–8 are *supplementary postulates* that have to do with minimality of change.

Postulates for revision [AGM85]:

(K$*$1) If B is a belief set and ϕ is a sentence, then $B * \phi$ is a belief set (closure)

(K$*$2) $\phi \in B * \phi$ (success)

(K$*$3) $B * \phi \subseteq B + \phi$ (expansion)

(K$*$4) If $\neg\phi \notin B$, then $B + \phi \subseteq B * \phi$ (preservation)

(K$*$5) $B * \phi = \bot$ (the inconsistent belief set) if and only if $\vdash \neg\phi$ (consistency)

(K$*$6) If $\vdash \phi_1 \leftrightarrow \phi_2$ then $B * \phi_1 = B * \phi_2$ (equivalence)

(K$*$7) $B * (\phi_1 \wedge \phi_2) \subseteq (B * \phi_1) + \phi_2$

(K$*$8) If $\neg\phi_2 \notin B * \phi_1$, then $(B * \phi_1) + \phi_2 \subseteq B * (\phi_1 \wedge \phi_2)$

Postulates for contraction [AGM85]:

(K–1) If B is a belief set and ϕ is a sentence, then $B - \phi$ is a belief set (closure)

(K–2) $B - \phi \subseteq B$ (inclusion)

(K–3) If $\phi \notin B$, then $B - \phi = B$ (vacuity)

(K–4) If not $\vdash \phi$, then $\phi \notin B - \phi$ (success)

(K–5) $B \subseteq (B - \phi) + \phi$ (recovery)

(K–6) If $\vdash \phi_1 \leftrightarrow \phi_2$ then $B - \phi_1 = B - \phi_2$ (equivalence)

(K–7) $(B - \phi_1) \cap (B - \phi_2) \subseteq B - (\phi_1 \wedge \phi_2)$ (conjunctive overlap)

(K–8) If $\phi_1 \notin B - (\phi_1 \wedge \phi_2)$, then $B - (\phi_1 \wedge \phi_2) \subseteq B - \phi_2$ (conjunctive conclusion)

Now, let us dissect the rationale of each postulate and "translate" into our setting. Let u be an update on a triple store $G = \mathcal{A}_G \cup \mathcal{T}_G$. Further, let $\mathcal{A}$ be an ABox, $\mathcal{T}$ a TBox, Sem an update semantics and $[\![\cdot]\!]_G$ be the SPARQL-semantics as per [PAG09] (or, Def. 19). In the following we will define different update semantics for SPARQL Update, which we will denote as $\mathbf{Sem}_x$ where the subscript x denotes a particular semantics. Moreover, we will use superscripts mat and red to denote semantics defined for materialised, or resp., reduced triple stores. In particular, $\mathbf{Sem}^{mat}_{naïve}$ stands for the naïve update semantics applied to a materialised triple store, whereas $\mathbf{Sem}^{red}_{naïve}$ stands for naïve update semantics applied to a reduced store. Furthermore, with $\mathbf{Sem}^{mat}_{\ldots}$ we assume any semantics that evaluates $[\![\cdot]\!]_G$ *without* rewriting, whereas any semantics $\mathbf{Sem}^{red}_{\ldots}$ evaluates $[\![\cdot]\!]_G$ *with* rewriting. In other words, in the former case, the WHERE clause is evaluated as usual, whereas in the latter case the WHERE clause is evaluated using the rewriting (i.e., the equivalent of being a-priori materialised in RDFS). For the definition of G^{sem}_u refer to Def. 24.

In the following, we define postulates for SPARQL based on the AGM postulates, by also grouping the original postulates whenever possible using '+' symbol.

K1 = (K∗1) + (K–1) rationale: the result of an update semantics sem, given an update u, is either a materialised or a reduced triple store G. Formally, $G = mat(G) \Rightarrow G^{sem}_u = mat(G^{sem}_u)$, or $G = red(G) \Rightarrow G^{sem}_u = red(G^{sem}_u)$.

K∗2 rationale: insertions should result in the inserted triples being true in the updated triple store. If $u = \texttt{INSERT}\{\mathcal{A}\}$ then $[\![\mathcal{A}]\!]_{G^{sem}_u} = \{\emptyset\}$[1].

K–2 rationale: a deletion should not add triples. If $u = \texttt{DELETE}\{\mathcal{A}\}$ then $G^{sem}_u \subseteq G$.

K∗3 rationale: the result of an insertion of any materialise- resp. reduce-preserving update semantics should be contained by the expansion operator. For our purposes, we need to distinguish the intended behaviour here, depending on whether we talk about materialise-preserving or reduce-preserving semantics, i.e., we write G^{expand}_u here short for $mat(G \cup \mathcal{A})$ for materialised-preserving semantics, and $red(G \cup \mathcal{A})$ for reduce-preserving semantics. With this auxiliary notation, we can define K∗3 as follows:

$u = \texttt{INSERT}\{\mathcal{A}\} \Rightarrow [\![?S\ ?P\ ?O]\!]_{G^{sem}_u} \subseteq [\![?S\ ?P\ ?O]\!]_{G^{expand}_u}$.

K–3 rationale: the deletion of triples not present in the G should not have an effect. If $u = \texttt{DELETE}\{\mathcal{A}\}$ and $[\![\mathcal{A}]\!]_G = \emptyset$[1] then $G^{sem}_u = G$.

K∗4 rationale: if there are no inconsistencies with respect to the new update, the result of any materialise- resp. reduce-preserving update semantics should contain the expansion operator. If $u = \texttt{INSERT}\{\mathcal{A}\}$ and $mat(G \cup \mathcal{A}) \not\models \bot \Rightarrow [\![?S\ ?P\ ?O]\!]_{G^{expand}_u} \subseteq [\![?S\ ?P\ ?O]\!]_{G^{sem}_u}$.

[1] Note that SPARQL boolean queries in the sense of Def. 2 can be defined in terms of an empty solution set {} meaning $\bot$, whereas—for a ground query without variables—a single empty binding $\{\emptyset\}$, means $\top$.

K–4 rationale: deletions should result in the deleted triples no longer be true in the updated triple store. If $u = \texttt{DELETE}\{\mathcal{A}\}$ then $[\![\mathcal{A}]\!]_{G_u^{sem}} = \emptyset$.

K∗5 rationale: if the update is inconsistent, then it should have no effect. For K∗5, in our context as we will see, it makes sense to discuss different variants of K∗5 separately, depending whether the inserted data $\mathcal{A}$ is inconsistent with the terminological data $\mathcal{T}_G \subseteq G$ (we will call such updates intrinsically inconsistent) or whether it is inconsistent with the TBox and the ABox data that is already present in G before the update, i.e., we define K∗5 and K∗5′ as follows.

K∗5: if $u = \texttt{INSERT}\{\mathcal{A}\}$ and $mat(\mathcal{A} \cup \mathcal{T}_G) \models \bot$ then $G_u^{sem} = G$.

K∗5′: if $u = \texttt{INSERT}\{\mathcal{A}\}$ and $mat(\mathcal{A} \cup G) \models \bot$ then $G_u^{sem} = G$.

K–5 rationale: delete followed by the equivalent insert should not lose triples. Formally, $(G^{sem}_{\texttt{DELETE}\{\mathcal{A}\}})^{sem}_{\texttt{INSERT}\{\mathcal{A}\}} \supseteq G$. For K-5, in our context, it makes sense to additionally discuss different variations of K–5′-K–5‴. This is due to the order of delete and inserts, as well as of different semantics, where both as combination might yield different results.

K–5′ rationale: delete followed by the equivalent insert results with no effect. Formally, if $\mathcal{A} \subseteq G$ then $(G^{sem}_{\texttt{DELETE}\{\mathcal{A}\}})^{sem}_{\texttt{INSERT}\{\mathcal{A}\}} = G$.

K–5″ rationale: insert followed by the equivalent delete should not lose triples. Formally, $(G^{sem}_{\texttt{INSERT}\{\mathcal{A}\}})^{sem}_{\texttt{DELETE}\{\mathcal{A}\}} \supseteq G$.

K–5‴ rationale: insert of new triples followed by the equivalent delete results with no effect. Formally, if $\mathcal{A} \cap mat(G) = \emptyset$ then $(G^{sem}_{\texttt{INSERT}\{\mathcal{A}\}})^{sem}_{\texttt{DELETE}\{\mathcal{A}\}} = G$.

K6 = (K∗6) + (K–6) rationale: if two ABoxes $\mathcal{A}_1$ and $\mathcal{A}_2$ entail the same triples w.r.t. the TBox of G, then their update should have the same effects[2]. If $u_1 = \texttt{INSERT}\{\mathcal{A}_1\}$, $u_2 = \texttt{INSERT}\{\mathcal{A}_2\}$, or $u_1 = \texttt{DELETE}\{\mathcal{A}_1\}$, $u_2 = \texttt{DELETE}\{\mathcal{A}_2\}$ and $mat(\mathcal{T}_G \cup \mathcal{A}_1) = mat(\mathcal{T}_G \cup \mathcal{A}_2)$ ($red(\mathcal{T}_G \cup \mathcal{A}_1) = red(\mathcal{T}_G \cup \mathcal{A}_2)$ resp.) then $G_{u_1}^{sem} = G_{u_2}^{sem}$.

In Chapter 4 and Chapter 5, we are going to rely on these postulates in order to check the rationality of various update semantics for SPARQL.

In order to make postulates applicable for TBox updates as well, we will merely replace ABox updates $\mathcal{A}$ with TBox updates $\mathcal{T}$, namely replace $u = \texttt{INSERT}\{\mathcal{A}\}$ and $u = \texttt{DELETE}\{\mathcal{A}\}$ with $u = \texttt{INSERT}\{\mathcal{T}\}$ and $u = \texttt{DELETE}\{\mathcal{T}\}$ respectively, where $\mathcal{A}$ and $\mathcal{T}$ stand for triples representing ABox or TBox statements, respectively as per Def. 16.

[2]K6 is known as the *syntax irrelevance postulate* which in other words states that in the revision process the syntax does not play a role, but rather the content that is represented by the syntax.

CHAPTER 4

Updating RDFS ABoxes and TBoxes in SPARQL

In this chapter we investigate alternative semantics for updates that preserve either materialised or reduced ABoxes, and discuss how these semantics can—similar to query answering—be implemented in off-the-shelf SPARQL 1.1 triple stores. Referring to Fig. 11, we will be talking about materialise- and reduce-preserving semantics for RDFS in terms of both ABox and TBox updates.

The SPARQL 1.1 Update [GPP13] and SPARQL/Entailment Regime [GOH+13] specification leaves it open how SPARQL endpoints should treat entailment regimes other than simple entailment in the context of updates. As a consequence, OBDM systems can not deal with implicit updates (cf. Ex. 21, reposted here):

Example 25 (Re-posted Ex. 21) Given the ABox and TBox data as in Ex. 3, the following SPARQL update operation tries to delete implied triples and to (re-)insert implied triple, based on the instantiations on the WHERE clause that might also instantiate implied triples:

```
DELETE { ?X a :Employee. }
INSERT { ?Y a :Department. }
WHERE { ?X :belongsTo ?Y. }
```

■

As discussed in Sec. 1.3, the main issue of updates under entailments is how updates shall deal with implied statements:

- What does it mean if an implied triple is explicitly (re-)inserted (or, resp., deleted)?

- Which (if any) additional triples should be inserted, (or, resp., deleted) upon updates?

In this chapter, we address such questions with the focus on a deliberately minimal ontology language, namely the minimal RDFS fragment of Muñoz et al. [MPG07] [1].

As it turns out, even in this confined setting, updates as defined in the SPARQL 1.1 Update specification impose non-trivial challenges; in particular, specific issues arise through the interplay of INSERT, DELETE, and WHERE clauses within a single SPARQL update operation, which—to the best of our knowledge—have not yet been considered in this combination in previous literature on updates under entailment (such as for instance [GHV11, CKNZ10]).

Existing triple stores offer different solutions to these problems, ranging from ignoring entailments in updates altogether, to keeping explicit and implicit (materialised) triples separate and re-materialising upon updates. In the former case (ignoring entailments) updates only refer to explicitly asserted triples, which may result in non-intuitive behaviours, whereas the latter case (re-materialisation) may be very costly, while still not eliminating all non-intuitive cases, as we will see. The problem is aggravated by the lack of a systematic approach to the question of which implied triples to store explicitly in a triple store and which not.

In this chapter we try to argue for a more systematic approach for dealing with updates in the context of RDFS entailments. More specifically, we will distinguish between two kinds of triple stores, that is *(*i) *materialised RDF stores*, which store all entailed ABox triples explicitly, and *(*ii) *reduced RDF Stores*, that is, redundancy-free RDF stores that do not store any assertional (ABox) triples already entailed by others. We propose alternative update semantics that preserve the respective types *(i)* and *(ii)* of triple stores, and discuss possible implementation strategies, partially inspired by query rewriting techniques from Ontology-Based Data Management (OBDM) [KRMZ13] and *DL-Lite* [CDGL$^+$07]. As already shown in [GHV11], erasure of ABox statements is deterministic in the context of RDFS, but insertion and particularly the interplay of DELETE/INSERT in SPARQL 1.1 Update has not been considered therein. Finally, we relax the initial assumption that terminological statements (using the RDFS vocabulary) are static, and discuss the issues that arise when also TBox statements are subject to updates (see Sec. 4.5).

4.1 *DL-Lite*$_{\text{rdfs}}$ Setting

In Chapter 2 we have given the necessary definitions about materialised and reduced triple stores respectively, namely, triple stores that store all implicit triples in an explicit

[1] We ignore issues like axiomatic triples [Hay04], blank nodes [MAHP11], or, in the context of OWL, inconsistencies arising through updates (see Chapter 5). Neither do we consider named graphs in SPARQL, which is why we talk about "triple stores" as opposed to "graph stores" [GPP13].

way, and triple stores that compute the core, i.e., do not store any implicit (redundant) triples respectively.

In the following, we provide a more formal definition for both types of triple stores, by first introducing the materialise and reduce operations respectively.

Definition 27 (Materialisation) *Given a triple store $G = \mathcal{T}_G \cup \mathcal{A}_G$ and inference rules for RDFS as defined in Fig. 23—which can be similarly represented as Datalog rules in Fig. 24—the* materialisation *$mat(G)$ is defined by exhaustively applying the four leftmost inference rules to G until a fix-point is reached, i.e., no new RDF (ABox) triple can be derived.*

We will call a triple store, or an ABox *materialised* if in each state it is always materialised.

Definition 28 (Materialised triple store) *We say that a triple store G or ABox is* materialised *if $G = mat(G)$.*

In the case of reduced triple stores, we note that this core [PPSW13] is uniquely determined in our setting whenever $\mathcal{T}$ is acyclic (which is usually a safe assumption)[2]; it could be naïvely computed by exhaustively "marking" each triple that can be inferred from applying any of the four leftmost rules in Fig. 23, and subsequently removing all marked elements of $\mathcal{A}$. An implementation of $red(G)$ using SPARQL/Update and property paths is provided in Sec. 7.1.

Definition 29 (Reduction) *Given a triple store $G = \mathcal{T}_G \cup \mathcal{A}_G$ and inference rules for RDFS as defined in Fig. 23—which can be similarly represented as Datalog rules in Fig. 24—the* reduction *$red(G)$ is defined by exhaustively applying the four leftmost inference rules to G by subsequently removing the implicit triples until no implicit triple is left in the triple store.*

Similarly, we will call a triple store, or an ABox *reduced* if in each state it is always reduced.

Definition 30 (Reduced triple store) *We say that a triple store G or ABox is* reduced *if $G = red(G)$.*

The following observation follows trivially.

[2]We note that even in the case when the TBox is cyclic we could define a deterministic way to remove redundancies, e.g., by preserving within a cycle only the lexicographically smallest ABox statements. That is, given TBox $A \sqsubseteq B \sqsubseteq C \sqsubseteq A$ and ABox $A(x), C(x)$; we would delete $C(x)$ and retain $A(x)$ only, to preserve reducedness.

Proposition 4 *Let $G = \mathcal{T}_G \cup \mathcal{A}_G$ be a triple store, q a CQ, and $\mathcal{A}'_G$ the set of ABox assertions in $mat(red(G))$. Then, $ans(q, G) = ans(rewrite(q, \mathcal{T}_G), red(\mathcal{A}_G)) = ans(q, \mathcal{A}'_G)$.*

Note that this proposition follows directly from Prop. 2, where instead of G we have $red(G)$. In other words, it says that query answers (under entailment) can either be obtained by re-materialisation of a reduced store, or alternatively, by query rewriting and posing it over a reduced store.

Lastly, we observe that, trivially, a triple store containing no ABox statements is both reduced and materialised.

Lemma 1 *Let $G = \mathcal{T}_G$ be a triple store with an empty ABox, then G is both reduced and materialised.*

Definition 31 (Mat-preserving and red-preserving semantics) *Let G and $u(P_d, P_i, P_w)$ be as in Def. 24. An update semantics Sem is called* mat-preserving*, if $G^{Sem}_{u(P_d,P_i,P_w)} = mat(G^{Sem}_{u(P_d,P_i,P_w)})$, and it is called* red-preserving*, if $G^{Sem}_{u(P_d,P_i,P_w)} = red(G^{Sem}_{u(P_d,P_i,P_w)})$.*

Specifically, we consider the following variants of materialised ABox preserving (or simply, *mat-preserving*) semantics and reduced ABox preserving (or simply, *red-preserving*) semantics, given an update $u(P_d, P_i, P_w)$:

Sem$_0^{mat}$: As a baseline for a mat-preserving semantics, we apply the naïve semantics (cf. Def. 24), followed by (re-)materialisation of the whole triple store.

Sem$_1^{mat}$: An alternative approach for a mat-preserving semantics is to follow the so-called "delete and rederive" algorithm [GMS93] for deletions, that is: (i) delete the instantiations of P_d *plus "dangling" effects*, i.e., effects of deleted triples that after deletion are not implied any longer *by any non-deleted triples*; (ii) insert the instantiations of P_i *plus all their effects*.

Sem$_2^{mat}$: Another mat-preserving semantics could take a different viewpoint with respect to deletions, following the intention to: (i) delete the instantiations of P_d *plus all their causes*; (ii) insert the instantiations of P_i *plus all their effects*.

Sem$_3^{mat}$: Finally, a mat-preserving semantics could combine **Sem**$_1^{mat}$ and **Sem**$_2^{mat}$, by deleting both causes of instantiations of P_d and their "dangling" effects.[3]

Sem$_0^{red}$: Again, the baseline for a red-preserving semantics would be to apply the naïve semantics, followed by (re-)reducing the triple store.

Sem$_1^{red}$: This red-preserving semantics extends **Sem**$_0^{red}$ by additionally deleting the causes of instantiations of P_d.

[3]Note the difference to the basic "delete and rederive" approach. **Sem**$_1^{mat}$ in combination with the intention of **Sem**$_2^{mat}$ would also mean to recursively delete effects of causes, and so forth. **Sem**$_3^{mat}$ as we have defined herein though, does not remove triples recursively.

The definitions of semantics $\mathbf{Sem}_0^{mat}$ and $\mathbf{Sem}_0^{red}$ are straightforward.

Definition 32 (Baseline mat-preserving and red-preserving update semantics) *Let G and $u(P_d, P_i, P_w)$ be as in Def. 24. Then, we define $\boldsymbol{Sem}_0^{mat}$ and $\boldsymbol{Sem}_0^{red}$ as follows:*

$$G^{\boldsymbol{Sem}_0^{mat}}_{u(P_d,P_i,P_w)} = mat(G_{u(P_d,P_i,P_w)}) \qquad G^{\boldsymbol{Sem}_0^{red}}_{u(P_d,P_i,P_w)} = red(G_{u(P_d,P_i,P_w)})$$

Let us proceed with a quick "reality-check" on these two baseline semantics by means of our running example.

Example 26 Consider the update from Ex. 21. It is easy to see that neither under $\mathbf{Sem}_0^{mat}$ executed on the materialised triple store of Ex. 15, nor under $\mathbf{Sem}_0^{red}$ executed on the reduced triple store of Ex. 16, it would have *any* effect. ∎

This behaviour is quite arguable. Hence, we proceed with discussing the implications of the proposed alternative update semantics, and how they could be implemented.

4.2 Alternative Mat-Preserving Semantics

We consider now in more detail different alternative mat-preserving semantics. As for $\mathbf{Sem}_1^{mat}$, we rely on a well-known technique in the area of updates for deductive databases called "delete and rederive" (DRed) [GMS93, CW94, VSM05, KBB11, UMJ+13] (also see Sec. 2.3). Informally translated to our setting, when given a logic program Π and its materialisation T_Π^ω, plus a set of facts $\mathcal{A}_d$ to be deleted and a set of facts $\mathcal{A}_i$ to be inserted, DRed (i) first deletes $\mathcal{A}_d$ and all its effects (computed via semi-naive evaluation [Ull88]) from T_Π^ω, resulting in $(T_\Pi^\omega)'$, (ii) then, starting from $(T_\Pi^\omega)'$, re-materialises $(T_\Pi^\omega)' \cup \mathcal{A}_i$ (again using semi-naive evaluation).

The basic intuition behind DRed of deleting effects of deleted triples and then re-materialising can be expressed in our notation as follows; as we will consider a variant of this semantics later on, we refer to this semantics as $\mathbf{Sem}_{1a}^{mat}$.

Definition 33 *Let $G = \mathcal{T}_G \cup \mathcal{A}_G$, $u(P_d, P_i, P_w)$, $\mathcal{A}_d$, and $\mathcal{A}_i$ be defined as in Def. 24. Then $G^{\boldsymbol{Sem}_{1a}^{mat}}_{u(P_d,P_i,P_w)} = mat(\mathcal{T}_G \cup (\mathcal{A}_G \setminus mat(\mathcal{T}_G \cup \mathcal{A}_d)) \cup \mathcal{A}_i)$.*

As opposed to the classic DRed algorithm, where Datalog distinguishes between view predicates (IDB) and extensional knowledge in the Database (EDB), in our setting we do not make this distinction, i.e., we do not distinguish between implicitly and explicitly inserted triples. This means that $\mathbf{Sem}_{1a}^{mat}$ would delete also those effects that had been inserted explicitly before.

We introduce now a different variant of this semantics, denoted $\mathbf{Sem}_{1b}^{mat}$, that makes a distinction between explicitly and implicitly inserted triples.

Definition 34 *Let $u(P_d, P_i, P_w)$ be an update operation, and $G = \mathcal{T}_G \cup \mathcal{A}_{expl} \cup \mathcal{A}_{impl}$ a triple store, where $\mathcal{A}_{expl}$ and $\mathcal{A}_{impl}$ respectively denote the explicit and implicit ABox triples. More formally, if we assume $\mathcal{A}_{impl} = \mathcal{A}_{expl} = \emptyset$ as the original state of an empty triple store, then the implicit and explicit ABox resulting from an update can be defined inductively as follows: $G^{\mathbf{Sem}^{mat}_{1b}}_{u(P_d,P_i,P_w)} = \mathcal{T}_G \cup \mathcal{A}'_{expl} \cup \mathcal{A}'_{impl}$, where $\mathcal{A}_d$ and $\mathcal{A}_i$ are defined as in Def. 24, with $\mathcal{A}'_{expl} = (\mathcal{A}_{expl} \setminus \mathcal{A}_d) \cup \mathcal{A}_i$, and $\mathcal{A}'_{impl} = mat(\mathcal{A}'_{expl} \cup \mathcal{T}_G) \setminus \mathcal{T}_G$.*

Note that in $\mathbf{Sem}^{mat}_{1b}$, as opposed to $\mathbf{Sem}^{mat}_{1a}$, we do not explicitly delete effects of $\mathcal{A}_d$ from the materialisation, since the definition just relies on re-materialisation from scratch from the explicit ABox $\mathcal{A}'_{expl}$. Nonetheless, the original DRed algorithm can still be used for computing $\mathbf{Sem}^{mat}_{1b}$ as shown by the following proposition.

Proposition 5 *Let us interpret the inference rules in Fig. 23 and triples in G respectively as rules and facts of a logic program Π; accordingly, we interpret $\mathcal{A}_d$ and $\mathcal{A}_i$ from Def. 34 as facts to be deleted from and inserted into Π, respectively. Then, the materialisation computed by DRed, as defined in [KBB11], computes exactly $\mathcal{A}'_{impl}$.*

None of $\mathbf{Sem}^{mat}_0$, $\mathbf{Sem}^{mat}_{1a}$, and $\mathbf{Sem}^{mat}_{1b}$ are equivalent, as shown in Ex. 27 below.

Example 27 Given the triple store $G = \{$`:C sc :D . :D sc :E`$\}$, on which we perform the operation `INSERT{:x a :C, :D, :E.}`, *explicitly* adding three triples, and subsequently perform `DELETE{:x a :C, :E.}`, we obtain, according to the three semantics discussed so far, the following ABoxes:
$\mathbf{Sem}^{mat}_0$: `{:x a :D. :x a :E.}` $\mathbf{Sem}^{mat}_{1a}$: `{}`
$\mathbf{Sem}^{mat}_{1b}$: `{:x a :D. :x a :E.}`

While after this update $\mathbf{Sem}^{mat}_0$ and $\mathbf{Sem}^{mat}_{1b}$ deliver the same result, the difference between these two is shown by the subsequent update `DELETE{:x a :D.}`
$\mathbf{Sem}^{mat}_0$: `{:x a :E.}` $\mathbf{Sem}^{mat}_{1a}$: `{}` $\mathbf{Sem}^{mat}_{1b}$: `{}` ■

As for the subtle difference between $\mathbf{Sem}^{mat}_{1a}$ and $\mathbf{Sem}^{mat}_{1b}$, we point out that none of [KBB11, UMJ+13], who both refer to using DRed in the course of RDF updates, make it clear whether explicit and implicit ABox triples are to be treated differently.

Further, continuing with Ex. 26, the update from Ex. 21 still would not have *any* effect, neither using $\mathbf{Sem}^{mat}_{1a}$, nor $\mathbf{Sem}^{mat}_{1b}$. That is, it is not possible in any of these update semantics to remove implicit information (without explicitly removing all its causes).

$\mathbf{Sem}^{mat}_2$ aims at addressing this problem concerning the deletion of implicit information. As it turns out, while the intention of $\mathbf{Sem}^{mat}_2$ to delete causes of deletions cannot be captured just with the *mat* operator, it can be achieved fairly straightforwardly, building upon ideas similar to those used in query rewriting.

As we have seen (cf. Ex. 14), in the setting of RDFS we can use Alg. 2.1 *rewrite* to expand a CQ to a UCQ that incorporates all its "causes". A slight variation can be used to compute the set of all causes, that is, in the most naïve fashion by just "flattening" the set of sets returned by Alg. 2.1 to a simple set; we denote this flattening operation on a set S of sets as $\mathit{flatten}(S)$. Let us illustrate it via an example.

Example 28 Given q and $\mathcal{T}$ as in Ex. 6. The example therein describes Alg. 2.1 applied to the query in the context of DL-Lite ontology language. As discussed previously, it can be used to compute the causes, though in the context of the RDFS ontology language, the "Reduce" step is not applicable. Hence, we have $S = \mathit{rewrite}(q, \mathcal{T})$: $S = \{\{worksFor(x, z), belongsTo(y, z)\}, \{worksFor(x, z), worksFor(y, z)\}\}$. Then, we compute: $\mathit{flatten}(S) = \{worksFor(x, z), worksFor(y, z), belongsTo(y, z)\}$. ∎

Likewise, we can easily define a modified version of $mat(G)$, applied to a BGP P using a TBox $\mathcal{T}$. This could be viewed as simply applying the first four left-most inference rules in Fig. 23 exhaustively to $P \cup \mathcal{T}$, and then removing $\mathcal{T}$. Let us call the resulting algorithm $mat_{\text{eff}}(P, \mathcal{T})$[4]. Using these considerations, we can thus define both rewritings that consider all causes, and rewritings that consider all effects of a given (insert or delete) pattern P:

Definition 35 (Cause/Effect rewriting) *Given a BGP P occurring as insert or delete pattern (i.e., $P = P_i$ or $P = P_d$) in an update operation over the triple store $G = \mathcal{T}_G \cup \mathcal{A}_G$, we define the* all-causes-rewriting *of P as $P^{caus} = \mathit{flatten}(\mathit{rewrite}(P, \mathcal{T}_G))$; likewise, we define the* all-effects-rewriting *of P as $P^{\mathit{eff}} = mat_{\text{eff}}(P, \mathcal{T}_G)$.*

This leads (almost) straightforwardly to a rewriting-based definition of $\mathbf{Sem}_2^{mat}$.

Definition 36 *Let $u(P_d, P_i, P_w)$ be an update operation. Then*

$$G^{\mathbf{Sem}_2^{mat}}_{u(P_d,P_i,P_w)} = G_{u(P_d^{caus},P_i^{\mathit{eff}},\{P_w\}\{P_d^{fvars}\})},$$

where $P_d^{fvars} = \{?v \text{ a rdfs:Resource.} \mid ?v \in Var(P_d^{caus}) \setminus Var(P_d)\}$.

The only tricky part in this definition is the rewriting of the WHERE clause, where P_w is joined[5] with a new pattern P_d^{fvars} that binds "anonymous" variables (i.e., the "fresh" variables denoted by '_' in Table 21, introduced by Alg. 2.1) in the rewritten DELETE clause, P_d^{caus}. Here, `?v a rdfs:Resource.` is a shortcut for a pattern which binds `?v` to any term occurring in G, cf. Sec. 7.1 for further details on how we implement this in practice.

[4] Note that the intention is not to provide optimised algorithms here, but just to convey the feasibility of this rewriting.

[5] A sequence of '{}'-delimited patterns in SPARQL corresponds to a join, where such joins can again be nested with UNIONs, with the obvious semantics, for details cf. Def. 18.

Example 29 Getting back to the materialised version of the triple store G from Ex. 15, the update u from Ex. 21 would, according to $\mathbf{Sem}_2^{mat}$, be rewritten to

```
DELETE {?X a :Employee. ?X :worksFor ?x1.}
INSERT {?Y a :Department . ?Y a :Organisation . }
WHERE  {{?X :belongsTo ?Y.}    {?x1 a rdfs:Resource.}}
```

with $G_u^{\mathbf{Sem}_2^{mat}}$ containing G without the triples `:john a :Employee; :worksFor :marketing.  :joe a :Employee; :worksFor :finance. :anna a :Employee; :worksFor :marketing; :worksFor :finance.  .` ■

It is easy to show that $\mathbf{Sem}_2^{mat}$ is mat-preserving. However, this semantics might still result in potentially non-intuitive behaviours. For instance, subsequent calls of INSERTs and DELETEs might leave "traces" in the form of "dangling" effects that remain after deletions, as shown by the following example.

Example 30 Assume G from Ex. 3 with an empty ABox. Under $\mathbf{Sem}_2^{mat}$, the following sequence of updates would leave as a trace `:joe a :Employee; :Person; :belongsTo :marketing .  :marketing a :Department; :Organisation .` , which would not be the case under the naïve semantics.

```
DELETE{} INSERT {:joe :worksFor :marketing.} WHERE{};
DELETE {:joe :worksFor :marketing.} INSERT{} WHERE{}
```

■

$\mathbf{Sem}_3^{mat}$ tries to address the issue of such "traces", but can no longer be formulated by a relatively straightforward rewriting. As regards the definition/implementation capturing the intention of $\mathbf{Sem}_3^{mat}$, there are two possible starting points, namely combining $\mathbf{Sem}_{1a}^{mat} + \mathbf{Sem}_2^{mat}$, or $\mathbf{Sem}_{1b}^{mat} + \mathbf{Sem}_2^{mat}$, respectively. In the following definition we intuitively build upon $\mathbf{Sem}_{1a}^{mat} + \mathbf{Sem}_2^{mat}$.

Definition 37 *Given $G = \mathcal{T}_G \cup \mathcal{A}_G$, $u(P_d, P_i, P_w)$, let $\mathcal{A}_d$, and $\mathcal{A}_i$ be defined as in Def. 24. Then, let $\mathcal{A}_d^{caus}$ be the instantiated ABox triples to be deleted plus their causes, and let $\mathcal{A}_i^{eff}$ be the instantiated ABox triples to be inserted plus their effects.*

Now we can define $\boldsymbol{Sem}_3^{mat}$:

$$G_{u(P_d,P_i,P_w)}^{\boldsymbol{Sem}_3^{mat}} = mat(\mathcal{T}_G \cup (\mathcal{A}_G \setminus \mathcal{A}_d^{caus} \setminus mat(\mathcal{T}_G \cup \mathcal{A}_d^{caus})) \cup \mathcal{A}_i^{eff}.$$

Let us illustrate $\mathbf{Sem}_3^{mat}$ via an example.

Example 31 Assume G and the sequence of updates from the previous example. Under $\mathbf{Sem}_3^{mat}$, the sequence of updates would not leave any trace, i.e., triples `:joe a :Employee; :Person; :belongsTo :marketing .  :marketing a :Department; :Organisation .` would be removed by the subsequent DELETE operation. ■

We emphasise though, that $\mathbf{Sem}_3^{mat}$ would still potentially run into arguable cases, since it might run into removing seemingly "disconnected" implicit assertions, whenever removed assertions cause these, as shown by the following example.

Example 32 Assume a materialised triple store G consisting only of the TBox triples `:Father sc :Person, :Male .` The behaviour of the following update sequence under a semantics implementing the intention of $\mathbf{Sem}_3^{mat}$ is arguable:

```
DELETE {} INSERT {:x a :Father.} WHERE {};
DELETE {:x a :Male.} INSERT {} WHERE {}
```

The result of the update sequence results with an empty ABox. We leave it open for now whether "recursive deletion of dangling effects" is intuitive: in this case, should upon deletion of x being Male, also the fact be deleted that x is a Person? ∎

In a strict reading of $\mathbf{Sem}_3^{mat}$'s intention, `:x a :Person.` would count as a dangling effect of the cause for `:x a :Male.`, since it is an effect of the inserted triple with no other causes in the store, and thus should be removed upon the delete operation.

Lastly, we point out that while implementations of (materialised) triple stores may make a distinction between implicit and explicitly inserted triples (e.g., by storing explicit and implicit triples separately, as sketched in $\mathbf{Sem}_{1b}^{mat}$ already), we consider the distinction between implicit triples and explicitly inserted ones non-trivial in the context of SPARQL 1.1 Update: for instance, is a triple inserted based upon implicit bindings in the WHERE clause of an INSERT statement to be considered "explicitly inserted" or not? We tend towards avoiding such distinction, and such philosophical aspects of possible SPARQL update semantics are left beyond the scope of this dissertation. For now, we turn our attention to the potential alternatives for red-preserving semantics.

4.3 Alternative Red-Preserving Semantics

Again, similar to $\mathbf{Sem}_3^{mat}$, for both baseline semantics $\mathbf{Sem}_0^{red}$ and $\mathbf{Sem}_1^{red}$ we leave it open whether they can be implemented by rewriting to SPARQL update operations following the naïve semantics, i.e., without the need to apply $red(G)$ over the whole triple store after each update; a strategy to avoid calling $red(G)$ would roughly include the following steps:

- delete the instantiations P_d *plus all the effects of instantiations* of P_i, which will be implied anyway upon the new insertion, thus preserving reduced;
- insert instantiations of P_i only if they are *not implied*, that is, they are not already implied by the current state of G or all their causes in G were to be deleted.

We leave further investigation of whether these steps can be cast into update requests directly by rewriting techniques to future work. Rather, we show that we can capture the intention of $\mathbf{Sem}_1^{red}$ by a straightforward extension of the baseline semantics.

Definition 38 ($\mathbf{Sem}_1^{red}$) *Let $u(P_d, P_i, P_w)$ be an update operation. Then*

$$G_{u(P_d,P_i,P_w)}^{\boldsymbol{Sem}_1^{red}} = red(G_{u(P_d^{caus},P_i,\{rewrite(P_w)\}\{P_d^{fvars}\})}),$$

where P_d^{caus} and P_d^{fvars} are as before.

Example 33 Getting back to the reduced version of the triple store G from Ex. 16, the update u from Ex. 21 would, according to $\mathbf{Sem}_1^{red}$, be rewritten to

```
DELETE { ?X a :Employee. ?X :worksFor ?x1. }
INSERT { ?Y a :Department. }
WHERE  {
         {
           { ?X :belongsTo ?Y. }
           UNION
           { ?X :worksFor ?Y. }
         }
         { ?x1 a rdfs:Resource.}
       }
```

with $G_u^{\mathbf{Sem}_1^{red}}$ containing the triples:
`:finance a :Department.  :marketing a :Department. .`

Observe here that the triple `:john :belongsTo :marketing.` cannot be entailed anymore because its causes are deleted, which some might view it as non-intuitive. ■

In a reduced store effects of P_d need not be deleted, which makes the considerations that led us to $\mathbf{Sem}_3^{mat}$ irrelevant for a red-preserving semantics, as shown next.

Example 34 Under $\mathbf{Sem}_1^{red}$, as opposed to $\mathbf{Sem}_2^{mat}$, the update sequence of Ex. 30 would leave no traces. However, the update sequence of Ex. 32 would likewise result in an empty ABox, again losing idempotence of single triple insertion followed by deletion. ■

Note that, while the rewriting for $\mathbf{Sem}_1^{red}$ is similar to that for $\mathbf{Sem}_2^{mat}$, post-processing for preserving reducedness is not available in off-the-shelf triple stores. Instead, $\mathbf{Sem}_2^{mat}$ could be readily executed by rewriting on existing triple stores, preserving materialisation. Again we refer to more details on possible implementations routes to Chapter 7.

4.4 Postulates for Mat-/Red-Preserving Semantics

For all the various semantics discussed, we check them against the postulates K1-K6 provided in Sec. 3.2, in order to see the rationality of the respective semantics. Table 41 gives a summary on all the semantics versus postulates. Speaking in general, $\mathbf{Sem}_{1b}^{mat}$ fulfils the highest number of postulates, whereas $\mathbf{Sem}_3^{mat}$ fulfils the least number of

	K1	K∗2	K–2	K∗3	K–3	K∗4	K–4	K∗5	K∗5′	K–5	K–5′	K–5″	K–5‴	K6	Total /14
$\mathbf{Sem}^{mat}_{naïve}$	X	✓	✓	✓	✓	X	✓	✓	✓	✓	✓	✓	✓	X	11
$\mathbf{Sem}^{red}_{naïve}$	X	✓	✓	✓	✓	✓	X	✓	✓	✓	✓	✓	✓	X	11
$\mathbf{Sem}^{mat}_{0}$	✓	✓	✓	✓	✓	✓	X	✓	✓	✓	X	✓	X	X	10
$\mathbf{Sem}^{mat}_{1a}$	✓	✓	✓	✓	X	✓	X	✓	✓	✓	✓	X	✓	✓	11
$\mathbf{Sem}^{mat}_{1b}$	✓	✓	✓	✓	✓	✓	X	✓	✓	✓	✓	✓	✓	X	12
$\mathbf{Sem}^{mat}_{2}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	X	X	X	X	9
$\mathbf{Sem}^{mat}_{3}$	✓	✓	✓	✓	X	✓	✓	✓	✓	X	X	X	X	X	8
$\mathbf{Sem}^{red}_{0}$	✓	✓	✓	✓	✓	✓	X	✓	✓	✓	✓	X	X	✓	11
$\mathbf{Sem}^{red}_{1}$	✓	✓	✓	✓	X	✓	✓	✓	✓	X	✓	X	X	X	9

Table 41: Checking postulates K1-K6 against $\mathbf{Sem}^{mat}_{x}$ and $\mathbf{Sem}^{red}_{x}$

postulates. In the following, for each one of them we give more explanation on why they are fulfilled or not fulfilled.

$\mathbf{Sem}^{mat}_{naïve}$ does not satisfy K1 due to lack of the materialise operator. It satisfies K∗2 because insertion of triples via the semantics of SPARQL/Update results with the triples being inserted. In fact, we note that for the same argument K∗2 holds in all the discussed semantics. It satisfies K–2 because a deletion of triples via the baseline SPARQL/Update semantics does not add new triples, and neither does it for any of the extensions/update semantics defined in this dissertation, so again this postulate holds for all the discussed update semantics[6]. It satisfies K∗3 because $G^{naive}_u = G \cup \mathcal{A} \subseteq mat(G \cup \mathcal{A}) = G^{expand}_u$, therefore $G^{naive}_u \subseteq G^{expand}_u$. It satisfies K–3 because deletion of a triple that does not exist in the triple store results with no changes. K∗4 is not satisfied, a trivial counter-example would be to `INSERT {:john a :Employee}` for $G =$`{:Employee sc :Person}`. It satisfies K–4 because deletion of any non-empty set of triples $\mathcal{A}$ via the semantics of SPARQL/Update results with the triples being deleted, and thus querying such triples results with an empty set. K∗5 and K∗5′ are trivially satisfied (again in all semantics mentioned in this chapter) because we only consider RDFS ontology language which does not have inconsistencies. K–5′ is satisfied because deletion and insertion of the same triples negates each other via SPARQL/update semantics. K–5 holds because K–5′ holds. K–5‴ is satisfied because insertion and deletion of the same triples negates each other via SPARQL/update semantics. K–5″ holds because K–5‴ holds. It does not satisfy K6 because if $u_1 =$ `DELETE{`$\mathcal{A}_1$`}`, $u_2 =$ `DELETE{`$\mathcal{A}_2$`}` such that $\mathcal{A}_1 \subset \mathcal{A}_2$ and $mat(\mathcal{T}_G \cup \mathcal{A}_1) = mat(\mathcal{T}_G \cup \mathcal{A}_2)$, then $G^{sem}_{u_1} \neq G^{sem}_{u_2}$; same holds for inserts. This is illustrated via an example.

Example 35 (K6 is not satisfied in $\mathbf{Sem}^{mat}_{naïve}$)
Let $G = \mathcal{T}_G \cup \mathcal{A}_G =$ {`:Employee sc :Person`}, $\mathcal{A}_1 =$ {`:john a :Employee`} and $\mathcal{A}_2 =$ {`:john a :Employee . :john a :Person .`}, such that $\mathcal{A}_1 \subset \mathcal{A}_2$ and $mat(\mathcal{T}_G \cup \mathcal{A}_1) =$

[6]Note that if we considered OPTIONAL patterns in P_w this might change, due to the non-monotonic flavour; however, we leave the consideration of such complex patterns beyond BGPs to future work.

$mat(\mathcal{T}_G \cup \mathcal{A}_2)$. Given $u_1 = \texttt{DELETE}\{\mathcal{A}_1\}$, $u_2 = \texttt{DELETE}\{\mathcal{A}_2\}$, then we conclude that $G_{u_1}^{\mathbf{Sem}_{naïve}^{mat}} \neq G_{u_2}^{\mathbf{Sem}_{naïve}^{mat}}$ holds. ■

$\mathbf{Sem}_{naïve}^{red}$ does not satisfy K1 due to lack of the reduce operator. Note though, that in case of DELETE only (i.e., if the INSERT is empty), K1 is satisfied. K∗2 and K–2 hold for the same reasons as for $\mathbf{Sem}_{naïve}^{mat}$. K∗3 is satisfied as $G_u^{naive} = G \cup \mathcal{A} \supseteq red(G \cup \mathcal{A}) = G_u^{expand}$, thus we have $[\![?S\ ?P\ ?O]\!]_{G_u^{naive}} = [\![?S\ ?P\ ?O]\!]_{G_u^{expand}}$, i.e., the query answering using query rewriting in this case amounts to same results. It satisfies K–3 because deletion of a triple that does not exist in the triple store results with no changes. K∗4 is satisfied because in K∗3 equality ("=") holds, and thus the other side follows (⇒). K–4 is not satisfied because if an implicit triple is deleted, then querying the triple with rewriting results with a non-empty set. K∗5, K∗5′, K–5, K–5′, K–5″ and K–5‴ hold for the same reasons as for $\mathbf{Sem}_{naïve}^{mat}$. It does not satisfy K6 because if $u_1 = \texttt{INSERT}\{\mathcal{A}_1\}$, $u_2 = \texttt{INSERT}\{\mathcal{A}_2\}$ such that $\mathcal{A}_1 \subset \mathcal{A}_2$ and $red(\mathcal{T}_G \cup \mathcal{A}_1) = red(\mathcal{T}_G \cup \mathcal{A}_2)$, then $G_{u_1}^{\mathbf{Sem}_{naïve}^{red}} \neq G_{u_2}^{\mathbf{Sem}_{naïve}^{red}}$; this is due to lack of reduce operator. Same as with Ex. 35, an analogous example can be constructed.

$\mathbf{Sem}_0^{mat}$ satisfies K1 because it relies upon the materialise operator. K∗2 and K–2 hold for the same reasons as for $\mathbf{Sem}_{naïve}^{mat}$. K∗3 is satisfied because equality ("=") holds, i.e., the semantics coincide by definition. K–3 is satisfied because deletion of a non-existing triple has no effect, consequently as well as materialise operator applied on that state. K∗4 is satisfied because equality ("=") holds, same as it holds in K∗3. It does not satisfy K–4 because deletion of triples does not guarantee that they are not going to be re-inserted after the materialise operator. K∗5 and K∗5′ are satisfied for the same reasons as for $\mathbf{Sem}_{naïve}^{mat}$. K–5′ and K–5‴ are not satisfied since deletions and insertions are not invertible due to the materialise operator, and thus consequently more triples could be added. Analogously, K–5 and K–5″ are satisfied. This semantics does not satisfy K6 because if $u_1 = \texttt{DELETE}\{\mathcal{A}_1\}$, $u_2 = \texttt{DELETE}\{\mathcal{A}_2\}$ such that $\mathcal{A}_1 \subset \mathcal{A}_2$ and $mat(\mathcal{T}_G \cup \mathcal{A}_1) = mat(\mathcal{T}_G \cup \mathcal{A}_2)$, then $G_{u_1}^{\mathbf{Sem}_0^{mat}} \neq G_{u_2}^{\mathbf{Sem}_0^{mat}}$ (see Ex. 35); note that for insertions it holds though.

$\mathbf{Sem}_{1a}^{mat}$ satisfies K1 for the same reason as $\mathbf{Sem}_0^{mat}$. K∗2 and K–2 hold for the same reasons as for $\mathbf{Sem}_{naïve}^{mat}$. K∗3 is satisfied because for $u = \texttt{INSERT}\{\mathcal{A}\}$ the semantics coincides with $\mathbf{Sem}_0^{mat}$. It does not satisfy K–3 because deletion of a non-existing triple may still result with deletion of other triples, namely triples having no alternative derivation. This is clarified by the following example:

Example 36 (K–3 is not satisfied in $\mathbf{Sem}_{1a}^{mat}$)
Assume $G = \mathcal{T}_G \cup \mathcal{A}_G$, such that $\mathcal{T}_G$ is given as in Ex. 3 and $\mathcal{A}_G = \{$`:john :belongsTo :finance`$\}$. Given the following update u:

```
DELETE {:john :worksFor :finance . }
```

according to $\mathbf{Sem}_{1a}^{mat}$, we get:

```
DELETE {:john :worksFor :finance . :john :belongsTo :finance .}
```

thus resulting with the triple `:john :belongsTo :finance` to be deleted as well. ■

Though, one could circumvent the issue by using the rewriting given in sequel.

Example 37 (cont'd)
Assume $G = \mathcal{T}_G \cup \mathcal{A}_G$ and u defined as in Ex. 36. Then, the following rewriting, pushes the triples in the DELETE template as "join" in the WHERE clause:

```
DELETE {  ?x :worksFor :finance . ?x :belongsTo :finance . }
WHERE  {  ?x :worksFor :finance . ?x :belongsTo :finance .
                    FILTER (?x = :john)
       }
```

With this rewriting, if the triple `:john :worksFor :finance` is not present in $\mathcal{A}_G$, the update is not going to delete `:john :belongsTo :finance`. ■

Fixing this issue seems non-trivial, but we leave it for future work to adapt the definition of the semantics accordingly.

K∗4 is satisfied for the same reason as in $\mathbf{Sem}_0^{mat}$. K–4 is not satisfied because despite deleting the triple and its effects, still it could be derived by the (re-)materialisation. K∗5 and K∗5′ hold for the same reasons as for $\mathbf{Sem}_{naive}^{mat}$. K–5′ is satisfied, due to being the inverse of K–5‴, thus triples can at least be implicitly recovered.

Example 38 (K–5′ is satisfied in $\mathbf{Sem}_{1a}^{mat}$)
Assume $G = \mathcal{T}_G \cup \mathcal{A}_G$, such that $\mathcal{T}_G$ is given as in Ex. 3 and $\mathcal{A}_G$ is empty. Given the following sequence of updates:

```
DELETE {} INSERT {:joe a :Person; a :Employee.} WHERE {};
DELETE {:joe a :Employee.} INSERT {} WHERE {};
DELETE {} INSERT {:joe a :Employee.} WHERE {};
```

According to $\mathbf{Sem}_{1a}^{mat}$, the last two operations negate each other, thus the triple `:joe a :Person` is recovered. ■

K–5 is satisfied because K–5′ is satisfied (and also the semantics is invertible for delete/insert). K–5″ is not satisfied because triples that are explicitly added could be potentially removed, and this is further clarified by the following example.

Example 39 (K–5″ is not satisfied in $\mathbf{Sem}_{1a}^{mat}$)
Assume $G = \mathcal{T}_G \cup \mathcal{A}_G$, such that $\mathcal{T}_G$ is given as in Ex. 3 and $\mathcal{A}_G$ is empty. Given the following sequence of updates:

```
DELETE {} INSERT {:joe a :Person.} WHERE {};
DELETE {} INSERT {:joe a :Employee.} WHERE {};
DELETE {:joe a :Employee.} INSERT {} WHERE {};
```

According to $\mathbf{Sem}_{1a}^{mat}$, the update results with the triple `:joe a :Person` additionally to be deleted. ■

K–5‴ is satisfied because given the restriction if $\mathcal{A} \cap mat(G) = \emptyset$ then we first insert new triples (that can't be inferred from other triples) plus the effects. Then, deletion removes all of them. Thus, for this semantics INS/DEL is invertible.

Finally, $\mathbf{Sem}_{1a}^{mat}$ satisfies K∗6 because for both deletes and inserts it relies upon the materialise operator, i.e., if $u_1 = \texttt{DELETE}\{\mathcal{A}_1\}$, $u_2 = \texttt{DELETE}\{\mathcal{A}_2\}$ such that $\mathcal{A}_1 \subset \mathcal{A}_2$ and $mat(\mathcal{T}_G \cup \mathcal{A}_1) = mat(\mathcal{T}_G \cup \mathcal{A}_2)$, then $G_{u_1}^{\mathbf{Sem}_{1a}^{mat}} = G_{u_2}^{\mathbf{Sem}_{1a}^{mat}}$; same holds for insertions as well.

$\mathbf{Sem}_{1b}^{mat}$ is similar to $\mathbf{Sem}_{1a}^{mat}$ with the difference that it satisfies K–3 and K–5″, but does not satisfy K6. It satisfies K–3 because the deletion of a triple that does not exist makes no difference in the explicit triple store ($\mathcal{A}_{expl}$), therefore resulting with no changes in the implicit store ($\mathcal{A}_{impl}$). It does not satisfy K–4: suppose $\mathcal{A}_{expl} = \{$`:x a :Employee; a :Person.`$\}$, then given $u =$ `DELETE{:x a :Person}` posed over $\mathcal{A}_{expl}$, it will be re-derived by the materialisation. It satisfies K–5‴ since updates are first executed over the explicit store, thus insertion followed by deletion of the same triples negates each other via SPARQL/Update semantics. K–5″ is satisfied because K–5‴ is satisfied. $\mathbf{Sem}_{1b}^{mat}$ does not satisfy K∗6 because if $u_1 = \texttt{INSERT}\{\mathcal{A}_1\}$, $u_2 = \texttt{INSERT}\{\mathcal{A}_2\}$ such that $\mathcal{A}_1 \subset \mathcal{A}_2$ and $mat(\mathcal{T} \cup \mathcal{A}_1) = mat(\mathcal{T} \cup \mathcal{A}_2)$, then $G_{u_1}^{\mathbf{Sem}_{1b}^{mat}} \neq G_{u_2}^{\mathbf{Sem}_{1b}^{mat}}$, in this context means that the respective explicit stores would be different; the same holds for deletes.

$\mathbf{Sem}_{2}^{mat}$ satisfies K1 for the same reason as $\mathbf{Sem}_{0}^{mat}$. K∗2 and K–2 hold for the same reasons as for $\mathbf{Sem}_{naïve}^{mat}$. K∗3 and K∗4 are satisfied for the same reasons as for $\mathbf{Sem}_{1a}^{mat}$. It satisfies K–3 because deletion of a non-existing triple, plus its causes results with no changes. That is, if causes would exist in $mat(G)$ then also the triple would exist in $mat(G)$. It satisfies K–4 because according to the definition, triples plus their causes are deleted from the materialised triple store. K∗5 and K∗5′ are satisfied for the same reason as for $\mathbf{Sem}_{naïve}^{mat}$. K–5, K–5′, K–5″, K–5‴ are not satisfied because insertion would result with triples ought to be inserted plus effects, whereas deletion would result with triples ought to be deleted plus the causes, where the causes are disjoint from effects, see Ex. 30. $\mathbf{Sem}_{2}^{mat}$ does not satisfy K6 because if $\mathcal{A}_1 \subset \mathcal{A}_2$ and $mat(\mathcal{T}_G \cup \mathcal{A}_1) = mat(\mathcal{T}_G \cup \mathcal{A}_2)$, then $G_{u_1}^{\mathbf{Sem}_{2}^{mat}} \neq G_{u_2}^{\mathbf{Sem}_{2}^{mat}}$. This is clarified by the following example.

Example 40 (K6 is not satisfied in $\mathbf{Sem}_{2}^{mat}$)
Consider $G = \mathcal{T}_G \cup \mathcal{A}_G$, such that $\mathcal{T}_G$ is given as in Ex. 3 and $\mathcal{A}_G$ is empty. Given ABoxes $\mathcal{A}_1 = \{$`:john a :Employee.`$\}$, $\mathcal{A}_2 = \{$`:john a :Employee; a :Person .`$\}$, and the corresponding updates $u_1 = \texttt{DELETE}\{\mathcal{A}_1\}$ $u_2 = \texttt{DELETE}\{\mathcal{A}_2\}$, such that $\mathcal{A}_1 \subset \mathcal{A}_2$ and $mat(\mathcal{T}_G \cup \mathcal{A}_1) = mat(\mathcal{T}_G \cup \mathcal{A}_2)$, then clearly $G_{u_1}^{\mathbf{Sem}_{2}^{mat}} \neq G_{u_2}^{\mathbf{Sem}_{2}^{mat}}$. Nonetheless, for inserts it holds though, for $mat(\mathcal{T}_G \cup \mathcal{A}_1) = mat(\mathcal{T}_G \cup \mathcal{A}_2)$ then $G_{u_1}^{\mathbf{Sem}_{2}^{mat}} = G_{u_2}^{\mathbf{Sem}_{2}^{mat}}$, since it relies upon "effects", which have the same outcome as the materialise operator. ■

$\mathbf{Sem}_3^{mat}$ is similar to $\mathbf{Sem}_2^{mat}$ with the difference that it does not satisfy K–3. It does not satisfy K–3 because the deletion of a triple that does not exist in the triple store, plus its causes, and effects of causes results with more triples being deleted.

Example 41 (K–3 not satisfied in $\mathbf{Sem}_3^{mat}$)
Assume $G = \mathcal{T}_G \cup \mathcal{A}_G$, such that $\mathcal{T}_G$ is given as in Ex. 3 and $\mathcal{A}_G = \{$`:john a :Person`$\}$. Given the following update:

```
DELETE {:john :belongsTo :finance . }
```

according to $\mathbf{Sem}_3^{mat}$, we get:

```
DELETE {:john :belongsTo :finance ; :worksFor :finance; a :Employee ;
        a :Person . :finance a :Department; a :Organization .}
```

The final update results with the triple `:john a :Person` to be deleted. ■

Now let us continue the discussion on postulate satisfiability with the reduce-preserving semantics.

$\mathbf{Sem}_0^{red}$ satisfies K1 because relies upon the reduce operator. It satisfies K$*$2 because the insertion of triples via the semantics of SPARQL/Update results with the triples being inserted. Even if removed by the reduce operator, querying for the triples would take the rewriting into account. It satisfies K–2 because the deletion of triples via the semantics of SPARQL/Update, plus reduce operator, do not add new triples. K$*$3 and K$*$4 are satisfied for the same reasons as for $\mathbf{Sem}_{1a}^{mat}$. It satisfies K–3 because the deletion of a non-existing triple in a reduced triple store results with no changes. K–4 is not satisfied because triples plus their causes are not deleted. K$*$5 and K$*5'$ are satisfied for the same reasons as in $\mathbf{Sem}_{naïve}^{mat}$. K–5$'$ is satisfied because in a reduced state, the deletion and insertion of the same triples negate each other via SPARQL/update semantics. K–5 holds because K–5$'$ holds. K–5$''$ and K–5$'''$ are not satisfied, because inserts followed by deletes could lose triples, this is further explained by the following example.

Example 42 (K–5$''$ and K–5$'''$ are not satisfied in $\mathbf{Sem}_0^{red}$)
Assume $G = \mathcal{T}_G \cup \mathcal{A}_G$, such that $\mathcal{T}_G$ is given as in Ex. 3 and $\mathcal{A}_G = \{$`:joe a :Person`$\}$. Given the following update:

```
DELETE {} INSERT {:joe a :Employee .} WHERE {};
DELETE {:joe a :Employee .} INSERT {} WHERE {};
```

The updates would result with the triple `:joe a :Person` being deleted. ■

It satisfies K6 because for either deletes or inserts u_1 and u_2, we have $red(\mathcal{T}_G \cup \mathcal{A}_1) = red(\mathcal{T}_G \cup \mathcal{A}_2)$ then directly holds $G_{u_1}^{\mathbf{Sem}_0^{red}} = G_{u_2}^{\mathbf{Sem}_0^{red}}$; this is due to the semantics relying upon the reduce operator.

$\mathbf{Sem}_1^{red}$ different from $\mathbf{Sem}_0^{red}$, it does not satisfy K–3, K–5, K6, whereas it satisfies K–4. K–3 is not satisfied because deleting a triple that does not exist in the reduced triple store might still delete as it also removes the causes; note that this does not occur in $\mathbf{Sem}_2^{mat}$ because of materialisation. K–4 is satisfied because according to the definition, triples plus their causes are removed from the reduced triple store. It does not satisfy K–5, further explained by the example.

Example 43 (K–5 is not satisfied in $\mathbf{Sem}_1^{red}$)
Assume $G = \mathcal{T}_G \cup \mathcal{A}_G$, such that $\mathcal{T}_G$ is given as in Ex. 3 and $\mathcal{A}_G = \{$`:joe a :Employee`$\}$. Given the following sequence of updates:

```
DELETE {:joe a :Person .} INSERT {} WHERE {};
DELETE {} INSERT {:joe a :Person .} WHERE {};
```

Then, we would also lose `:joe a :Employee .` due to deleting triples plus their causes. ■

Similar to $\mathbf{Sem}_0^{red}$, K–5′ is satisfied because in a reduced state and also given the constraint for the update if $\mathcal{A} \subseteq G$ (otherwise Ex. 43 would apply and thus consequently would not be satisfied), then the deletion and insertion of the same triples negate each other via SPARQL/update semantics.

It does not satisfy K6, and this is clarified in the following example.

Example 44 (K6 is not satisfied in $\mathbf{Sem}_1^{red}$)
Assume $G = \mathcal{T}_G \cup \mathcal{A}_G$, which is already in the reduced state $red(G) = \{$`:joe a :Employee .:joe a :Father . :Father sc :Person . :Employee sc :Person .`$\}$, and the deletions consist of the ABox-es:
$\mathcal{A}_1 = \{$`:joe a :Employee; a :Person .`$\}$ and $\mathcal{A}_2 = \{$`:joe a :Employee .`$\}$ such that $red(\mathcal{T}_G \cup \mathcal{A}_1) = red(\mathcal{T}_G \cup \mathcal{A}_2)$, then we get $G_{u_1}^{\mathbf{Sem}_1^{red}} \neq G_{u_2}^{\mathbf{Sem}_1^{red}}$. This is because deleting `:joe a :Person .` would also delete `:joe a :Father .`. ■

4.5 TBox Updates

So far, we have considered the TBox as static. As already noted in [GHV11], additionally allowing TBox updates considerably complicates issues and opens additional degrees of freedom for possible semantics. While in this dissertation we do not explore all of these, we limit ourselves to sketch these different degrees of freedom and suggest one pragmatic approach to extend updates expressed in SPARQL to RDFS TBoxes.

In order to allow for TBox updates, we have to extend the update language: in the following, we will assume *general BGPs*, extending Def. 17.

Definition 39 (general BGP) *A* general BGP *is a set of triples of any of the forms from Table 22, where* $x, y, A, A', P, P' \in \Gamma \cup \mathcal{V}$.

Note that we need to restrict the use of general BGPs in updates if we want to avoid meta-reasoning or resulting with non-standard RDF(S). To this end, we define a general update operation as follows:

Definition 40 (general update) *A* general update *is an update $u(P_d, P_i, P_w)$ where we add the following additional restrictions: each variable in P_d, P_i, P_w is used either in only class positions, only property positions or only individual positions.*

We observe that with this relaxation for BGPs, updates as per Def. 23 can query TBox data, since they admit TBox triples in P_w. In order to address this issue we need to also generalise the definition of query answers.[7]

Definition 41 *Let Q be a union of general BGPs and $[\![Q]\!]_G$ the simple SPARQL semantics as per [PAG09](or, Def. 19), i.e., essentially the set of query answers obtained as the union of answers from simple pattern matching of the general BGPs in Q over the graph G. Then we define $ans_{\mathsf{RDFS}}(Q, G) = [\![Q]\!]_{mat(G)}$.*

In fact, Def. 41 does not affect ABox inferences, that is, the following corollary follows immediately from Prop. 2 for non-general UCQs as per Def. 17.

Corollary 1 *Let Q be a UCQ as per Def. 17. Then $ans_{\mathsf{RDFS}}(Q, G) = ans_{\mathsf{rdfs}}(Q, G)$.*

As opposed to the setting discussed so far, where the right-most two rules in Fig. 23 used for TBox materialisation were ignored, we now focus on the discussion of terminological updates under the standard "intensional" semantics, cf. [FGM+13] (essentially defined by the inference rules in Fig. 23) and attempt to define a reasonable (that means computable) semantics under this setting. Note that upon terminological queries, the RDFS semantics and DL semantics differ, since this "intensional" semantics does not cover all terminological inferences derivable in DL, the details of this aspect are beyond the scope of the dissertation.

Observation 1. TBox updates potentially affect both materialisation and reducedness of the ABox, that is, (i) upon TBox *insertions* a materialised ABox might need to be re-materialised in order to preserve materialisation, and, respectively, a reduced ABox might no longer be reduced; (ii) upon TBox *deletions* in a materialised setting, we have a similar issue to what we called "dangling" effects earlier, whereas in a reduced setting indirect deletions of implied triples could cause unintuitive behaviour.

Let us illustrate the last point of reduced setting via an example, for the materialised setting see Ex. 49; the other points are rather self-explanatory.

[7] As mentioned in Fn. 9, elements of Γ may act as individuals, concept, or roles names in parallel.

Example 45 If we are given the graph $G = \{$`:A sc :B. :B sc :C.`$\}$. If we pose the following insert $u =$ `INSERT` $\{$:x *a* :A . :x *a* :C$\}$, which then followed by reduce operation we get $G = \{$`:x a :A . :A sc :B. :B sc :C.`$\}$. If we pose $u =$ `DELETE` $\{$:x *a* :A$\}$, we lose the triple `:x a :C.`, which was previously added explicitly. Likewise, we lose the triple if we pose $u =$ `DELETE` $\{$:A *sc* :C$\}$ instead of previous update. For some use cases this might be viewed as non-intuitive.

■

Observation 2. Whereas deletions of implicit ABox triples can be achieved deterministically by deleting all single causes, TBox deletions involving `sc` and `sp` chains can be achieved in several distinct ways, as already observed by [GHV11].

Example 46 Consider the graph $G = \{$`:A sc :B. :B sc :C. :B sc :D. :C sc :E. :D sc :E. :E sc :F.`$\}$ with the update `DELETE{:A sc :F.}`

Independent of whether we assume a materialised TBox, we would have various choices here to remove triples, to delete all the causes for `:A sc :F`. ■

In order to define a deterministic semantics for TBox updates, we need a canonical way to delete implicit and explicit TBox triples. Minimal cuts are suggested in [GHV11] in the `sc` (or `sp`, resp.) graphs as candidates for deletions of `sc` (or `sp`, resp.) triples. The problem of minimal multicut in graph theory is to find a minimal set of edges in a graph, such that their removal disconnects the pairs of vertices provided as input to a given graph. However, as easily verified by Ex. 46, minimal multicuts are still ambiguous.

Here, we suggest two update semantics using rewritings to SPARQL 1.1 property path patterns [HS13] that yield canonical minimal cuts.

Definition 42 *Let $u(P_d, P_i, P_w)$ be an update operation as per Def. 40 where P_d, P_i, P_w are general BGPs. Then*

$$G^{\boldsymbol{Sem}^{mat}_{outcut}}_{u(P_d,P_i,P_w)} = mat(G_{u(P'_d,P_i,P'_w)}),$$

where each triple $\{A_1\ sp\ A_2\} \in P_d$ such that $sp \in \{\mathsf{sc}, \mathsf{sp}\}$ is replaced within P'_d by $\{A_1\ sp\ ?x.\}$, and we add to P'_w the property path pattern $\{A_1\ sp\ ?x.\ ?x\ sp^*\ A_2\}$. Analogously, $\boldsymbol{Sem}^{mat}_{incut}$ is defined by replacing $\{?x\ sp\ A_2\}$ within P'_d, and adding $\{A_1\ sp^*\ ?x.\ ?x\ sp\ A_2\}$ within P'_w instead.

Both $\mathbf{Sem}^{mat}_{outcut}$ and $\mathbf{Sem}^{mat}_{incut}$ may be viewed as straightforward extensions of $\mathbf{Sem}^{mat}_{0}$, i.e., both are mat-preserving and equivalent to the baseline semantics for non-general BGPs (i.e., on ABox updates):

Proposition 6 *Let $u(P_d, P_i, P_w)$ be an update operation, where P_d, P_i, P_w are (non-general) BGPs. Then*

$$G^{\boldsymbol{Sem}^{mat}_{outcut}}_{u(P_d,P_i,P_w)} \quad = \quad G^{\boldsymbol{Sem}^{mat}_{incut}}_{u(P_d,P_i,P_w)} \quad = \quad G^{\boldsymbol{Sem}^{mat}_{0}}_{u(P_d,P_i,P_w)}.$$

The intuition behind the rewriting in $\mathbf{Sem}^{mat}_{outcut}$ is to delete for every deleted *A sp B.* triple, all directly outgoing *sp* edges from *A* that lead into paths to *B*, or, resp., in $\mathbf{Sem}^{mat}_{incut}$ all directly incoming edges to *B*. The intuition to choose these canonical minimal cuts is motivated by the following proposition.

Proposition 7 *Let $u =$ DELETE $\{A\ sp\ B\}$, and G a triple store with materialised TBox $\mathcal{T}_G$. Then, the TBox statements deleted by $G^{\boldsymbol{Sem}^{mat}_{outcut}}_{u(P_d,P_i,P_w)}$ (or, $G^{\boldsymbol{Sem}^{mat}_{incut}}_{u(P_d,P_i,P_w)}$, resp.) form a minimal cut [GHV11] of $\mathcal{T}_G$ disconnecting A and B.*

Proof 1 (Sketch) *In a materialised TBox, one can reach B from A either directly or via n direct neighbours $C_i \neq B$, which (in)directly connect to B. So, a minimal cut contains either the multicut between A and the C_is, or between the C_is and B; the latter multicut requires at least the same amount of edges to be deleted as the former, which in turn corresponds to the outbound cut. This proves the claim for $\boldsymbol{Sem}^{mat}_{outcut}$. We can proceed analogously for $\boldsymbol{Sem}^{mat}_{incut}$.* ∎

The following example illustrates that the generalisation of Prop. 7 to updates involving the deletion of several TBox statements at once does not hold.

Example 47 Assume the materialised triple store $G = \{$`:A` *sp* `:B, :C, :D.` `:B` *sp* `:C, :D.`$\}$ and $u =$ `DELETE{:A` *sp* `:C.` `:A` *sp* `:D.}`. Here, $\mathbf{Sem}^{mat}_{incut}$ does not yield a minimal *multicut* in G w.r.t. disconnecting (`:A, :C`) and (`:A, :D`), as it deletes more triples `{:A` *sp* `:C, :D.` `:B` *sp* `:C, :D.}` whereas minimal *multicut* would yield `{:A` *sp* `:B, :C, :D.}` ∎

Now, let us construct a similar example for $\mathbf{Sem}^{mat}_{outcut}$.

Example 48 Assume the materialised triple store $G = \{$`:B` *sp* `:A, :C, :D.` `:C` *sp* `:A, :D.` `:D` *sp* `:A.`$\}$ and $u =$ `DELETE{:B` *sp* `:A.` `:C` *sp* `:A.}`. Here, $\mathbf{Sem}^{mat}_{outcut}$ does not yield a minimal *multicut* in G w.r.t. disconnecting (`:B, :A`) and (`:C, :A`), as it deletes more triples `{:B` *sp* `:A, :C, :D.` `:C` *sp* `:A, :D.}` whereas minimal *multicut* would yield `{:B` *sp* `:A, :C` *sp* `:A, :D` *sp* `:A.}` ∎

We can combine ABox + TBox semantics independently within a single update by just combining the respective rewritings from both ABox and TBox semantics. For instance, if we combine $\mathbf{Sem}_2^{mat}$ for ABox updates with $\mathbf{Sem}_{incut}^{mat}$ for TBox updates, which would be a viable option given that both delete implicit triples and insert effects. Let us illustrate this via an example.

Example 49 Getting back to the materialised version of the triple store G from Ex. 15, plus including the TBox materialisation, and the following update u:

```
DELETE { ?X a :Employee. :worksFor sp :belongsTo .}
INSERT { ?Y a :Department . }
WHERE  { ?X :belongsTo ?Y . }
```

according to $\mathbf{Sem}_2^{mat}$ and $\mathbf{Sem}_{incut}^{mat}$ would be rewritten to:

```
DELETE {?X a :Employee. ?X :worksFor ?x1. ?Z sp :belongsTo.}
INSERT {?Y a :Department . ?Y a :Organisation . }
WHERE  {{ {?X :belongsTo ?Y.}    {?x1 a rdfs:Resource.}}
       UNION { :worksFor sp* ?Z . ?Z sp :belongsTo . }}
```

followed by a *mat* operation as required by $\mathbf{Sem}_{incut}^{mat}$. Observe that we have separated ABox and TBox rewritings by using a UNION in the WHERE clause. This is due to the fact that in-existence of the TBox triple ought to be deleted should not affect the ABox triples (if it were to be "join" using '.') and vice versa. Note that deleting `:worksFor sp :belongsTo.` would leave as "dangling" effects all resources having `:belongsTo` properties, which are derived from resources having `worksFor` properties (this general update removes such resources as seen from its ABox part). This behavior of resulting with dangling effects after TBox update is also earlier stressed in *Observation 1.* ■

Note that in the previous example we are dealing with ABox+TBox updates in separation without any interleaving, i.e., TBox semantics does not affect the ABox semantics and vice versa.

As the examples show, the extension of the baseline ABox update semantics to TBox updates already yields new degrees of freedom. We leave a more in-depth discussion of TBox updates also extending the other semantics from Sec. 4.2 for future work.

4.6 Postulates for Mat-Preserving TBox Semantics

Same as with ABox semantics, we check the postulates against the TBox semantics $\mathbf{Sem}_{incut}^{mat}$ and $\mathbf{Sem}_{outcut}^{mat}$. As mentioned in Sec. 3.2, the same postulates apply by merely replacing the ABox update $\mathcal{A}$ with the TBox update $\mathcal{T}$.

In Table 42 is given an overview of the TBox semantics we defined versus the postulates. Both semantics fulfill the same postulates, not surprisingly, given that they are very similar. They both delete an implicit TBox triple albeit with a different outcome –

as regards to the triples being deleted. We also provide in Table 42 the postulate satisfiability for (i) non-general BGPs, i.e., ABox updates, (ii) general BGPs, but in case of TBox updates only, and (iii) general BGPs in case of both ABox and TBox updates. We can see that based on this order, i.e., if we use the semantics for ABox updates we have the most postulates satisfied (10), for TBox updates only we have one less (9), and if we use the semantics for both ABox + TBox updates then we have the least number of postulates satisfied (8).

In the following we give more explanation on why the semantics fulfill or not fulfill the postulates.

$\mathbf{Sem}^{mat}_{incut}$ and $\mathbf{Sem}^{mat}_{outcut}$ satisfy K1 because they rely upon the materialise operator. They satisfy K$*$2 because the insertion of triples via the semantics of SPARQL/Update results with the triples being inserted, and in this case it boils down to the merge of graphs followed by the materialise operator. They satisfy K–2 because the deletion of triples via the semantics of SPARQL/Update does not add new triples, also both semantics in the case of deletions do not trigger insertions. K$*$3 is satisfied because for $u = \texttt{INSERT}\{\mathcal{A}\}$ the semantics coincide with $\mathbf{Sem}^{mat}_{expand}$. They satisfy K–3 because deletion of a non-existing triple results with no changes. That is, if we delete a non-existing triple `:A sc :B` it means that there is no path from the starting node `:A` to the ending node `:B`, consequently according to the definitions of both semantics there will be no other triples to be deleted. K$*$4 is satisfied because equality ("=") holds as in K$*$3. They satisfy K–4 because according to the definition, the triple ought to be deleted and all other triples that entail this triple are removed from the materialised triple store. K$*$5 and K$*5'$ are trivially satisfied because we only consider RDFS which does not have inconsistencies. K–5, K–5′, K–5″ and K–5‴ are not satisfied because insertion may result with triples to be inserted plus effects, whereas deletion may result with triples to be deleted plus other triples (as provided in Def. 42), which are disjoint from the effects. In the following we give examples using $\mathbf{Sem}^{mat}_{incut}$, but in analogous way we can construct examples for $\mathbf{Sem}^{mat}_{outcut}$.

Example 50 (K–5 and K–5′ are not satisfied in $\mathbf{Sem}^{mat}_{incut}$)
Consider $G = \mathcal{T}_G = mat(G) = \{$ `:A` *sc* `:B .` `:B` *sc* `:C .` `:A` *sc* `:C .`$\}$, $\mathcal{T} = \{$ `:A` *sc* `:C`$\}$, $u_1 = \texttt{DELETE}\{\mathcal{T}\}$ and $u_2 = \texttt{INSERT}\{\mathcal{T}\}$. Then,

$$(G_{u_1}^{\mathbf{Sem}^{mat}_{incut}})_{u_2}^{\mathbf{Sem}^{mat}_{incut}} = \{\texttt{:A } sc \texttt{ :B . :A } sc \texttt{ :C .}\} \not\supseteq G$$

This proves that K–5 and K–5′ are not satisfied. ∎

Next, we also prove why K–5″ is not satisfied via an example.

Example 51 (K–5″ is not satisfied in $\mathbf{Sem}^{mat}_{incut}$)
Consider $G = \mathcal{T}_G = mat(G) = \{$ `:A` *sc* `:B .` `:B` *sc* `:C .` `:A` *sc* `:C .`$\}$.

Now, consider $\mathcal{T} = \{$`:A` *sc* `:C`$\}$, $u_1 = \texttt{DELETE}\{\mathcal{T}\}$ and $u_2 = \texttt{INSERT}\{\mathcal{T}\}$. Then,

$$(G_{u_2}^{\mathbf{Sem}_{incut}^{mat}})_{u_1}^{\mathbf{Sem}_{incut}^{mat}} = \{\texttt{:A } sc \texttt{ :B}\} \not\supseteq G$$

■

Now, we prove why K-5‴ is not satisfied. Note that the restriction $\mathcal{T} \cap mat(G) = \emptyset$ means that we will be inserting and removing an explicit triple. According to the definitions of the semantics, after insert (of an explicit triple) we have *mat* operator which might add new implicit triples, followed by delete (of an explicit triple) that removes the explicit triple. As side-effect we have implicit triples residing as "dangling" effects in the triple store. This proves that we are nonetheless adding more triples in the triple store, and this holds for both TBox semantics.

Example 52 (K–5‴ is not satisfied in $\mathbf{Sem}_{incut}^{mat}$)
Consider $G = \mathcal{T}_G = mat(G) = \{\texttt{:A } sc \texttt{ :B . :B } sc \texttt{ :C . :A } sc \texttt{ :C .}\}$.

Now, consider $\mathcal{T} = \{\texttt{:B } sc \texttt{ :D}\}$ (note that $\mathcal{T} \cap mat(G) = \emptyset$), $u_1 = \texttt{DELETE}\{\mathcal{T}\}$ and $u_2 = \texttt{INSERT}\{\mathcal{T}\}$. Then,

$$(G_{u_2}^{\mathbf{Sem}_{incut}^{mat}})_{u_1}^{\mathbf{Sem}_{incut}^{mat}} = \{\texttt{:A } sc \texttt{ :B . :B } sc \texttt{ :C . :A } sc \texttt{ :C . :A } sc \texttt{ :D }\} \supset G$$

Hence, $(G_{u_2}^{\mathbf{Sem}_{incut}^{mat}})_{u_1}^{\mathbf{Sem}_{incut}^{mat}} \neq G$.

For $\mathbf{Sem}_{outcut}^{mat}$ we get the very same result.

■

They do not satisfy K6, this is further explained by the following example.

Example 53 (K6 is not satisfied in $\mathbf{Sem}_{incut}^{mat}$)
Consider $G = \mathcal{T}_G = \emptyset$. Let TBoxes $\mathcal{T}_1 = \{$ `:A` sc `:B, :B` sc `:C, :B` sc `:D, :C` sc `:E, :D` sc `:E`$\}$ and $\mathcal{T}_2 = \mathcal{T}_1 \cup \{$ `:A` sc `:C, :A` sc `:D` $\}$ with the corresponding updates $u_1 = \texttt{DELETE}\{\mathcal{T}_1\}$, $u_2 = \texttt{DELETE}\{\mathcal{T}_2\}$, such that $\mathcal{T}_1 \subset \mathcal{T}_2$ and $mat(\mathcal{T}_G \cup \mathcal{T}_1) = mat(\mathcal{T}_G \cup \mathcal{T}_2)$. Then clearly $G_{u_1}^{\mathbf{Sem}_{incut}^{mat}} \neq G_{u_2}^{\mathbf{Sem}_{incut}^{mat}}$ holds.

Note that when deleting the implicit triple `:A` sc `:E` in $\mathcal{T}_2$, $\mathbf{Sem}_{incut}^{mat}$ would also delete, in addition, the triple `:A` sc `:D`, which is not in $\mathcal{T}_1$. ■

In analogous way, we can construct an example for $\mathbf{Sem}_{outcut}^{mat}$.

Note that from Table 42, the difference between both $\mathbf{Sem}_{incut}^{mat}$ and $\mathbf{Sem}_{outcut}^{mat}$ (TBox updates) versus $\mathbf{Sem}_{0}^{mat}$ is that in the former ones K–5 and K–5″ are not satisfied, whereas K–4 is satisfied. This is because $(G^{sem}_{\texttt{DELETE}\{\mathcal{A}\}})^{sem}_{\texttt{INSERT}\{\mathcal{A}\}} \not\supseteq G$ and $(G^{sem}_{\texttt{INSERT}\{\mathcal{A}\}})^{sem}_{\texttt{DELETE}\{\mathcal{A}\}} \not\supseteq G$ hold for both $\mathbf{Sem}_{incut}^{mat}$ and $\mathbf{Sem}_{outcut}^{mat}$, given that we have a rewriting for P_d which is not the case in $\mathbf{Sem}_{0}^{mat}$. In other words, $\mathbf{Sem}_{0}^{mat}$ does not remove more triples than provided in the original template P_d. But, this is the reason why it does not satisfy

K–4, as opposed to $\mathbf{Sem}^{mat}_{incut}$ and $\mathbf{Sem}^{mat}_{outcut}$ that both satisfy it. On the other hand, for non-general BGPs as outlined in Prop. 6 the semantics coincide, thus the postulate satisfaction of $\mathbf{Sem}^{mat}_{0}$ apply, cf. first and second row (ABox) in Table 42. In case of general BGPs where both ABox plus TBox is taken into account, then both semantics distinguish and process separately ABox and TBox updates. In other words, for ABox updates they perform same as $\mathbf{Sem}^{mat}_{0}$, whereas for TBox updates they do as provided in Def. 42. For that reason, for ABox+TBox updates the intersection of postulate satisfiability for ABox and TBox is computed in Table 42.

In the next chapter, we are going to see the challenges when we extend the expressivity of ontology with `owl:disjointWith` axioms. Such axioms can lead to inconsistencies in knowledge bases, and for that reason we devise other semantics tailored to deal with inconsistencies inspired by belief revision.

	K1	K∗2	K–2	K∗3	K–3	K∗4	K–4	K∗5	K∗5′	K–5	K–5′	K–5″	K–5‴	K6	Total /14
$\mathbf{Sem}^{mat}_{incut}$ (ABox)	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✗	✗	10
$\mathbf{Sem}^{mat}_{outcut}$ (ABox)	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✗	✗	10
$\mathbf{Sem}^{mat}_{incut}$ (TBox)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	9
$\mathbf{Sem}^{mat}_{outcut}$ (TBox)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	9
$\mathbf{Sem}^{mat}_{incut}$ (ABox+TBox)	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	8
$\mathbf{Sem}^{mat}_{outcut}$ (ABox+TBox)	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	8

Table 42: Checking postulates K1-K6 against $\mathbf{Sem}^{mat}_{incut}$ and $\mathbf{Sem}^{mat}_{outcut}$. We distinguish between non-general BGPs (ABox updates), general BGPs in case of TBox updates only, and general BGPs in case of both ABox + TBox updates.

CHAPTER 5

Resolving Inconsistencies using SPARQL

In the previous Chapter 4, we discussed several semantics of SPARQL updates in the context of RDFS ontologies and data, which altogether are stored in a triple store. We also distinguished the cases in which the triple store is fully materialised or to the contrary, is reduced to its minimal core so that all other triples can be derived using TBox axioms. In the case of materialised triple stores, we discussed both ABox and TBox updates, whereas for reduced resp. ABox updates only.

As a stepping stone, this chapter continues the study of SPARQL updates focusing on the role of *inconsistency* in supporting SPARQL ABox updates over materialised stores. As a minimalistic ontology language allowing for inconsistencies, we consider RDFS$_\neg$, an extension of RDFS [HPS14] with *class disjointness axioms* of the form $\{P$ `disjointWith` $Q\}$ from OWL [MGH$^+$12].

In the approaches proposed so far, either the update language is restricted to sets of ground atoms (see Sec. 8.1.1) or, where the full SPARQL update language is allowed, the TBox language is restricted so that no inconsistencies can arise (see Chapter 4). In this chapter we discuss directions to overcome these limitations based on ideas from belief revision.

Starting from a DL-Lite fragment covering RDFS and concept disjointness axioms, we define three semantics for SPARQL instance-level (ABox) update: under *cautious semantics*, inconsistencies are resolved by rejecting updates potentially introducing conflicts; under *brave semantics*, instead, conflicts are overridden in favor of new information where possible; finally, the *fainthearted semantics* is a compromise between the former two approaches, designed to accommodate as much of the new information as possible, as long as consistency with the prior knowledge is not violated (cf. Fig. 11). We show how these semantics can be implemented in SPARQL via rewritings of polynomial size

and draw first conclusions from their practical evaluation, which we discuss in a separate chapter (see Sec. 7.2).

The aim is to adapt the basic belief revision operators for efficient implementation of ABox updates expressed in SPARQL 1.1, in the presence of RDFS$_\neg$ TBox axioms.

Let us see challenges that arise in the new setting using an example.

Example 54 (Re-posted Ex. 22 and Ex. 23) Let TBox $\mathcal{T}_G$ consists of the following axioms:

```
:belongsTo rdfs:domain :Employee .
:belongsTo rdfs:range :Manager .
:Employee owl:disjointWith :Manager .
```

Consider the following SPARQL update request u in the context of the TBox $\mathcal{T}_G$:

```
INSERT {?X :belongsTo ?Y} WHERE  {?X :worksFor ?Y}
```

Consider an ABox with data on people that happen to work for each other, i.e., both being employee and employer: $\mathcal{A}_1 = \{$`:john :worksFor :anna.  :anna :worksFor :john`$\}$.

Here, u would create two assertions `:john :belongsTo :anna` and `:anna :belongsTo :john`. Due to the range and domain constraints in $\mathcal{T}$, these assertions result in clashes both for John and for Anna. Note that all inconsistencies are in the new data, and thus we say that u is *intrinsically inconsistent* for the particular ABox $\mathcal{A}_1$. We discuss how such updates can be fixed using SPARQL rewritings.

Now, let $\mathcal{A}_2$ be the ABox $\{$`:john :worksFor :anna.  :john a :Manager`$\}$. It is clear that after the update u, the ABox will become inconsistent with respect to $\mathcal{T}_G$ due to the property assertion `:john :belongsTo :anna`, implying that John is both a Employee and a Manager which contradicts the disjointness axiom. In contrast to the previous case, the clash here is between the prior knowledge and the new data. ■

Based on a semantics from Chapter 4, we propose *three update semantics* for this case, and provide *efficient SPARQL rewriting algorithms* for implementing them in the RDFS$_\neg$ setting.

Before we go into further detail, we initially define the problem setting and other necessary definitions that are relevant for the *DL-Lite*$_{\text{RDFS}_\neg}$ fragment.

5.1 *DL-Lite*$_{\text{rdfs}_\neg}$ Setting

We first introduce the *DL-Lite*$_{\text{RDFS}_\neg}$ ontology language, given that this the fragment we are going to use in this chapter.

	TBox	RDFS$_\neg$		TBox	RDFS$_\neg$		TBox	RDFS$_\neg$
1.	$A' \sqsubseteq A$	A' sc A.	3.	$\exists P \sqsubseteq A$	P dom A.	5.	$A' \sqsubseteq \neg A$	A' dw A.
2.	$P' \sqsubseteq P$	P' sp P.	4.	$\exists P^- \sqsubseteq A$	P rng A.			

	ABox	RDFS$_\neg$
6.	$A(x)$	x a A.
7.	$P(x, y)$	x P y.

Table 51: *DL-Lite*$_{\text{RDFS}_\neg}$ assertions vs. RDF(S), where A, A' denote concept (or, class) names, P, P' denote role (or, property) names, Γ is the set of IRI constants (excl. the OWL/RDF(S) vocabulary) and $x, y \in \Gamma$. For RDF(S), we use abbreviations (rsc, sp, dom, rng, a) as introduced in [MPG07].

Definition 43 (RDFS$_\neg$ ABox, TBox, triple store) *We call a set $\mathcal{T}_G$ of inclusion assertions of the forms 1–5 in Table 51 an* (RDFS$_\neg$) TBox*, a set $\mathcal{A}_G$ of assertions of the forms 6–7 in Table 51 an* (RDF) ABox*, and the union $G = \mathcal{T}_G \cup \mathcal{A}_G$ an* (RDFS$_\neg$) triple store.

As this fragment allows for inconsistencies, we need a definition for that.

Definition 44 (Consistency) *A triple store G is called consistent, if $mat(G)$ does not contain both $C(x)$ and $\neg C(x)$ for any concept C and constant $x \in \Gamma$.*

As already elaborated in Chapter 2, query answering in the presence of ontologies is done either by rule-based pre-materialisation of the ABox or by query rewriting. In the RDFS$_\neg$ case, materialisation in polynomial time is feasible. Let $mat(G)$ be the triple store obtained from exhaustive application of the inference rules in Fig. 51 on a consistent triple store G, and—analogously—let chase$(q, \mathcal{T}_G)$ refer to "materialisation" w.r.t. $\mathcal{T}_G$ applied to a CQ q. We call a triple store G (resp. the ABox of G) *materialised* if the equality $G \setminus \mathcal{T}_G = mat(G) \setminus \mathcal{T}_G$ holds. In this chapter, we will always focus on "materialisation preserving" semantics for SPARQL update operations, which we dubbed $\mathbf{Sem}_2^{mat}$ in Chapter 4 and which preserves a materialised triple store. We recall the

$$\frac{?C \text{ sc } ?D. \quad ?S \text{ a } ?C.}{?S \text{ a } ?D.} \qquad \frac{?P \text{ dom } ?C. \quad ?S\ ?P\ ?O.}{?S \text{ a } ?C.}$$

$$\frac{?P \text{ sp } ?Q. \quad ?S\ ?P\ ?O.}{?S\ ?Q\ ?O.} \qquad \frac{?P \text{ rng } ?C. \quad ?S\ ?P\ ?O.}{?O \text{ a } ?C.} \qquad \frac{?S \text{ a } ?C, ?D. \quad ?C \text{ dW } ?D.}{\bot}$$

Figure 51: Minimal RDFS rules from [MPG07] plus class disjointness "clash" rule from OWL2 RL [MGH+12].

intuition behind $\mathbf{Sem}_2^{mat}$, given an update $u = (P_d, P_i, P_w)$: (i) delete the instantiations of P_d *plus all their causes*; (ii) insert the instantiations of P_i *plus all their effects*.

Given an ABox assertion $\mathcal{E}$, $\mathcal{E}^{\text{caus}} = \{\mathcal{E}' \mid \mathcal{E} \in mat(\{\mathcal{E}'\} \cup \mathcal{T}_G)\}$. In the definition of $\mathcal{E}^{\text{caus}}$, if $\mathcal{E}$ is a class membership assertion `(x a C)`, then $\mathcal{E}'$ is one of `(x a C')`, `(x P ?Y)`, `(?Y P x)` for some fresh variable $?Y$, class `C'` and role `P`. If $\mathcal{E}$ is a role participation assertion `(x R z)`, $\mathcal{E}'$ is of the form `(x P z)`, for some role `P`.

Let us recall the definition of $\mathbf{Sem}_2^{mat}$, which we re-formulate using the notation elaborated above:

Definition 45 ($\mathbf{Sem}_2^{mat}$) *Let $u(P_d, P_i, P_w)$ be an update operation. Then*

$$G^{\boldsymbol{Sem}_2^{mat}}_{u(P_d,P_i,P_w)} = G_{u(P_d^{\text{caus}}, P_i^{\text{eff}}, \{P_w\}\{P_d^{fvars}\})}$$

Here, $P_d^{\text{caus}} = \bigcup_{\mathcal{E} \in atoms(P_d)} \mathcal{E}^{\text{caus}}$; $P_i^{\text{eff}} = \text{chase}(P_i, \mathcal{T}_G)$ and P_d^{fvars} is a pattern that binds variables occurring in P_d^{caus} but not in P_d to the constants from Γ occurring in G.

We refer to Chapter 4 for further details, but stress that as such, $\mathbf{Sem}_2^{mat}$ is not able to detect or deal with inconsistencies arising from extending G with instantiations of P_i. In what follows, we will discuss how this can be remedied.

Remark 1 *Note that although the DELETE clause P_d is syntactically a BGP, its semantics is different. Namely, triples occurring in P_d are mutually independent (cf. Def. 23), so that for every $\theta \in ans(P_w, G)$, each atom in $P_d\theta \cap G$ is deleted from G no matter which other atoms of $P_d\theta$ occur in G. Therefore, P_d^{caus} is computed atom-wise, unlike CQ rewriting [CDGL$^+$07]. Note that $|\mathcal{E}^{\text{caus}}| = O(||\mathcal{T}_G||)$ where $||\mathcal{T}_G||$ denotes the vocabulary size of $\mathcal{T}_G$: in each RDFS$_\neg$ derivation, i.e., derivation according to one of the rules in Fig. 51, a class membership assertion can occur at most once for each class in $\mathcal{T}_G$, and a role membership assertion can occur at most twice for every role in $\mathcal{T}_G$. Thus, $|P_d^{\text{caus}}| \leq 2|P_d| \cdot ||\mathcal{T}_G||$ and $|P_i^{\text{eff}}| \leq |P_i| \cdot ||\mathcal{T}_G||$, so both can be computed in poly-time. This underpins the polynomial complexity of our rewritings.*

5.2 Checking Consistency of a SPARQL Update

In the literature on the evolution of DL-Lite knowledge bases [CKNZ10, FKAC13], updates represented by pairs of ABoxes $\mathcal{A}_d, \mathcal{A}_i$ have been studied. However, whereas such update might be viewed to fit straightforwardly to the corresponding $\mathcal{A}_d, \mathcal{A}_i$ in Def. 23, it is typically assumed that $\mathcal{A}_i$ is consistent with the TBox, and thus one only needs to consider how to deal with inconsistencies between the update and the old state of the knowledge base. However, this a priori assumption may be insufficient for SPARQL updates, where concrete values for inserted triples are obtained from variable bindings in the WHERE clause, and depending on the bindings, the update can be either consistent or not. This is demonstrated by the update u from Ex. 54 which, when applied

Algorithm 5.1: Constructing a SPARQL ASK query to check intrinsic inconsistency (for the definition of P_i^{eff}, cf. Def. 36)

Input: RDFS$_\neg$ TBox $\mathcal{T}_G$, SPARQL update $u(P_d, P_i, P_w)$
Output: A SPARQL ASK query returning $True$ if u is intrinsically inconsistent

if $\bot \in P_i^{\text{eff}}$ **then**
  return ASK {} *//u contains clashes in itself, i.e., is inconsistent for any triple store*
end
else
  $W := \{\,\mathsf{FILTER}(\mathit{False})\}$; *//neutral element w.r.t. union*
  foreach *pair of triple patterns* $(?X\ \mathsf{a}\ A)$, $(?Y\ \mathsf{a}\ A')$ *in* P_i^{eff} **do**
    if $A \sqsubseteq \neg A' \in \mathcal{T}_G$ **then**
      $W := W\ \mathsf{UNION}\,\{\{P_w\theta_1[?X \mapsto ?Z]\}\,.\,\{P_w\theta_2[?Y \mapsto ?Z]\}\}$ for a fresh $?Z$
    end
  end
  return ASK WHERE $\{W\}$
end

to the ABox $\mathcal{A}_1$, results in an inconsistent set $\mathcal{A}_i$ of insertions. We call this *intrinsic inconsistency* of an update *relative to a triple store* $G = \mathcal{T}_G \cup \mathcal{A}_G$.

Definition 46 *Let G be a triple store. The update u is said to be* intrinsically consistent w.r.t. *G if the set of new assertions $\mathcal{A}_i$ from Def. 23 generated by applying u to G, taken in isolation from the ABox of G, does not contradict the TBox of G. Otherwise, the update is said to be intrinsically inconsistent w.r.t. G.*

Intrinsic inconsistency of the update differs crucially from the inconsistency w.r.t. the old state of the knowledge base, illustrated by the ABox $\mathcal{A}_2$ from Ex. 54. This latter case can be addressed by adopting an update policy that prefers newer assertions in case of conflicts, as studied in the context of DL-Lite KB evolutions [CKNZ10], which we will discuss in Sec. 5.3 below. Intrinsic inconsistencies however are harder to deal with, since there is no cue which assertion should be discarded in order to avoid the inconsistency. Our proposal here is thus to discard *all* mutually inconsistent pairs of insertions.

We first present an algorithm for checking intrinsic inconsistency by means of SPARQL ASK queries and then a safe rewriting algorithm. This rewriting is based on the observation that clashing triples can be introduced by a combination of two bindings of variables in the WHERE clause, as the Ex. 54 (the ABox $\mathcal{A}_1$) illustrates. To handle such cases, two copies of the WHERE clause P_w are created by the rewriting in Algorithms 5.1 and 5.2, for each pair of disjoint concepts according to the TBox of the triple store. These algorithms use the notation described in Rem. 2 below.

Algorithm 5.2: Safe rewriting safe(u)

Input: RDFS$_\neg$ TBox $\mathcal{T}_G$, SPARQL update $u(P_d, P_i, P_w)$
Output: SPARQL update safe(u)

```
1  if ⊥ ∈ P_i^eff then
2  |  return u(P_d, P_i, FILTER(False))
3  end
4  W := {FILTER(False)};    //neutral element w.r.t. union
5  foreach pair of triple patterns (?X a A), (?Y a A') in P_i^eff do
6  |  if A ⊑ ¬A' ∈ T_G then
7  |  |  //cf. Rem. 2 for notation θ[...]
8  |  |  W := W UNION {P_w θ1[?X ↦ ?Y]} UNION {P_w θ2[?Y ↦ ?X]}}
9  |  end
10 end
11 return u(P_d, P_i, P_w MINUS {W})
```

Remark 2 *Our rewriting algorithms rely on producing fresh copies of the WHERE clause. Assume θ, θ_1, θ_2, ... to be substitutions replacing each variable in a given formula with a distinct fresh one. For a substitution σ, we also define $\theta[\sigma]$ resp. $\theta_i[\sigma]$ to be an extension of σ, renaming each variable at positions not affected by σ with a distinct fresh one. For instance, let F be the triple ($?Z$ `:belongsTo` $?Y$). Now, $F\theta$ makes a variable disjoint copy of F: $?Z_1$ `:belongsTo` $?Y_1$ for fresh $?Z_1, ?Y_1$. $F[?Z \mapsto ?X]$ is just a substitution of $?Z$ by $?X$ in F. Finally, $F\theta[?Z \mapsto ?X]$ results in $?X$ `:belongsTo` $?Y_2$ for fresh $?Y_2$. We assume that all occurrences of $F\theta[\sigma]$ stand for syntactically the same query, but that $F\theta[\sigma_1]$ and $F\theta[\sigma_2]$, for distinct σ_1 and σ_2, can only have variables in $range(\sigma_1) \cap range(\sigma_2)$ in common.* ■

Using this notation, the possibility of unifying two variables $?X$ and $?Y$ in P_w on a given triple store can be tested with the query $\{P_w\theta_1[?X \mapsto ?Z]\}\{P_w\theta_2[?Y \mapsto ?Z]\}$ where θ_1 and θ_2 are variable renamings as in Rem. 2 and $?Z$ is a fresh variable.

In order to check the intrinsic consistency of an update, this condition should be evaluated for every pair of variables of P_w, the unification of which leads to a clash. A SPARQL ASK query based on this idea is produced by Alg. 5.1. Lines 1-3 of Alg. 5.1 check if the inserted data are inconsistent, i.e., if insert contains $\{?X$ a $A.$ $?X$ a $A'.\}$ such that A dw A' is in TBox $\mathcal{T}_G$, where in this case we drop the update. Note that it suffices to check only triples of the form $\{?X$ a $?C\}$ at line 5 of Alg. 5.1, since disjointness conditions can only be formulated for concepts, according to the syntax in Table 51. Furthermore, since we are taking the facts in P_i^{eff} extended by all facts implied by $\mathcal{T}_G$, at line 6 of Alg. 5.1 it suffices to check the disjointness conditions explicitly mentioned in $\mathcal{T}_G$ and not all those which are implied by $\mathcal{T}_G$. Note also that the DELETE clause P_d plays no role in this case, since we only consider clashes within inserted facts.

Example 55 Consider the update u from Ex. 54, in which the INSERT clause P_i can create clashing triples. To identify potential clashes, Alg. 5.1 first applies the inference rule for the range axiom, and computes $P_i^{\text{eff}} = \{?X \texttt{ a :Employee . } ?Y \texttt{ a :Manager}\}$. Now both variables $?X, ?Y$ occur in the triples of type (6) from Table 51 with clashing concept names. The following ASK query is produced by Alg. 5.1.

```
ASK WHERE { ?X :worksFor ?Y . ?Y :worksFor ?X1 }
```

(In this and subsequent examples we omit the trivial FILTER(*False*) union branch used in rewritings to initialize variables with disjunctive conditions, such as W in Alg. 5.1) ■

Suppose that an insert is not intrinsically consistent for a given triple store. One solution would be to discard it completely, should the above ASK query return *True*. Another option which we consider here is to only discard those variable bindings from the WHERE clause, which make the INSERT clause P_i inconsistent. This is the task of the *safe rewriting* safe(·) in Alg. 5.2, which removes all variable bindings that participate in a clash between different triples of P_i. Let P_w be a WHERE clause, in which the variables $?X$ and $?Y$ should not be unified to avoid clashes. With θ_1, θ_2 being "fresh" variable renamings as in Rem. 2, Alg. 5.2 uses the union of $P_w\theta_1[?X \mapsto ?Y]$ and $P_w\theta_2[?Y \mapsto ?X]$ to eliminate unsafe bindings that send $?X$ and $?Y$ to the same value.

Example 56 Alg. 5.2 extends the WHERE clause of the update u from Ex. 54 as follows:

```
INSERT{?X :belongsTo ?Y} WHERE{?X :worksFor ?Y
 MINUS{{?X1 :worksFor ?X} UNION {?Y :worksFor ?Y2}}}
```

Note that the safe rewriting can make the update void. For instance, safe(u) has no effect on the ABox $\mathcal{A}_1$ from Ex. 54, since there is no cue, which of `:john :worksFor :anna`, `:anna :worksFor :john` needs to be dismissed to avoid the clash. However, if we extend this ABox with assertions both satisfying the WHERE clause of u and not causing undesirable variable unifications, safe(u) would make insertions based on such bindings. For instance, adding the fact `:bob :worksFor :alice` to $\mathcal{A}_1$ would assert `:bob :belongsTo :alice` as a result of safe(u). ■

A rationale for using MINUS rather than FILTER NOT EXISTS in Alg. 5.2 (and also in a rewriting in forthcoming Sec. 5.3) can be illustrated by an update in which variables in the INSERT and DELETE clauses are bound in different branches of a UNION:

```
DELETE {?V a :Manager} INSERT {?X :belongsTo ?Y}
WHERE {{?X :worksFor ?Y} UNION {?V :worksFor ?W}}
```

A safe rewriting of this update is

```
DELETE {?V a :Manager} INSERT {?X :belongsTo ?Y}
WHERE { {{?X :worksFor ?Y} UNION {?V :worksFor ?W}}
        MINUS{ {{?X1 :worksFor ?X} UNION {?V1 :worksFor ?W1}}
         UNION {{?Y :worksFor ?Y2} UNION {?V2 :worksFor ?W2}} } }
```

It can be verified that with FILTER NOT EXISTS in place of MINUS this update makes no insertions on all triple stores: the branches `{?V1 :worksFor ?W1}` and `{?V2 :worksFor ?W2}` are satisfied whenever `{?X :worksFor ?Y}` is, making FILTER NOT EXISTS evaluate to *False* whenever `{?X :worksFor ?Y}` holds.

We conclude this section by formalizing the intuition of update safety. For a triple store G and an update $u = (P_d, P_i, P_w)$, let $[\![P_w]\!]^u_G$ denote the set of variable bindings computed by the query "$\mathsf{SELECT} ?X_1, \ldots, ?X_k \mathsf{WHERE}\ P_w$" over G, where $?X_1, \ldots, ?X_k$ are the variables occurring in P_i or in P_d.

Theorem 1 *Let $\mathcal{T}_G$ be a TBox, let u be a SPARQL update (P_i, P_d, P_w), and let query q_u and update* safe$(u) = (P_d, P_i, P'_w)$ *result from applying Alg. 5.1 resp. Alg. 5.2 to u and $\mathcal{T}_G$. Then, the following properties hold for an arbitrary RDFS$_\neg$ triple store $G = \mathcal{T}_G \cup \mathcal{A}_G$:*

(1) $q_u(G) = True$ *iff* $\exists \mu, \mu' \in [\![P_w]\!]^u_G$ *s.t.* $\mu(P_i) \wedge \mu'(P_i) \wedge \mathcal{T}_G \models \bot$;
(2) $[\![P_w]\!]^u_G \setminus [\![P'_w]\!]^u_G = \{\mu \in [\![P_w]\!]^u_G \mid \exists \mu' \in [\![P_w]\!]^u_G$ *s.t.* $\mu(P_i) \wedge \mu'(P_i) \wedge \mathcal{T}_G \models \bot\}$.

5.3 Materialisation Preserving Update Semantics

In this section we discuss resolution of inconsistencies between triples already in the triple store and newly inserted triples. Our baseline requirement for each update semantics is formulated as the following property.

Definition 47 (Consistency-preserving) *Let G be a triple store and $u(P_d, P_i, P_w)$ an update. A materialisation preserving update semantics Sem is called* consistency preserving *in RDFS$_\neg$ if the evaluation of update u, i.e., $G^{Sem}_{u(P_d,P_i,P_w)}$, results in a consistent triple store.*

Our consistency preserving semantics are respectively called *brave*, *cautious* and *faint-hearted*. The brave semantics always gives priority to newly inserted triples by discarding all pre-existing information that contradicts the update. The cautious semantics is exactly the opposite, discarding inserts that are inconsistent with facts already present in the triple store; i.e., the cautious semantics never deletes facts unless explicitly required by the DELETE clause of the SPARQL update. Finally, the fainthearted semantics executes the update partially, only performing insertions for those variable bindings which do not contradict existing knowledge (again, taking into account deletions).

All semantics rely upon incremental update semantics $\mathbf{Sem}^{mat}_2$, introduced in Chapter 4, which we aim to extend to take into account class disjointness. Note that for the present section we assume updates to be intrinsically consistent, which can be checked or enforced beforehand in a preprocessing step by the safe rewriting discussed in Sec. 5.2. In this section, we lift our definition of update operation to include also updates (P_d, P_i, P_w) with P_w produced by the safe rewriting Alg. 5.2 from some update satisfying Def. 23.

What remains to be defined is the handling of clashes between newly inserted triples and triples already present in the triple store.

The intuitions of our semantics for a SPARQL update $u(P_d, P_i, P_w)$ in the context of an RDFS$_\neg$ TBox are as follows:

- *brave semantics* $\mathbf{Sem}^{mat}_{brave}$: (i) delete all instantiations of P_d and their causes, *plus all the non-deleted triples in G clashing with instantiations of triples in P_i to be inserted*, again also including the causes of these triples; (ii) insert the instantiations of P_i plus all their effects.
- *cautious semantics* $\mathbf{Sem}^{mat}_{caut}$: (i) delete all instantiations of P_d and their causes; (ii) insert all instantiations of P_i plus all their effects, *unless they clash with some non-deleted triples in G*: in this latter case, do not perform the update.
- *fainthearted semantics* $\mathbf{Sem}^{mat}_{faint}$: (i) delete all instantiations of P_d and their causes; (ii) insert those instantiations of P_i (plus all their effects) which *do not clash with non-deleted triples in G*.

Remark 3 *Note that $\boldsymbol{Sem}^{mat}_2$ is not able to cope with so called "dangling" effects – that is, triples inserted at some point for the sake of materialisation, whose causes have been subsequently deleted. As pointed out in Chapter 4, one way to deal with this issue is to combine $\boldsymbol{Sem}^{mat}_2$ with marking of explicitly inserted triples. This approach was implemented as a semantics $\boldsymbol{Sem}^{mat}_{1b}$ in Chapter 4, splitting the ABox $\mathcal{A}$ into the explicit part $\mathcal{A}_{expl}$ and the implicit part $\mathcal{A}_{impl} = \mathcal{A} \setminus \mathcal{A}_{expl}$. $\mathcal{A}_{expl}$ can be maintained, e.g., in a separate RDF graph using a straightforward update rewriting. Now, deleting P_d would not only retract P_d^{caus} from $\mathcal{A}$, but also the triples in* $\text{chase}(P_d^{\text{caus}}, \mathcal{T}_G) \setminus \text{chase}(\mathcal{A}_{expl} \setminus P_d^{\text{caus}}, \mathcal{T}_G)$. *That is, the effects of P_d^{caus} are removed unless they can be derived from facts remaining in $\mathcal{A}$ after enforcing the deletion P_d. Such an aggressive removal of dangling triples can lead to counterintuitive behavior (cf. Ex. 32), and requires maintaining the explicit ABox $\mathcal{A}_{expl}$, which is why we opted to preserve dangling effects in our rewritings.*

We will now describe implementations of the three semantics above via SPARQL rewritings, which can be shown to be materialisation preserving and consistency preserving.

5.3.1 Brave Semantics

The rewriting in Alg. 5.3 implements the brave update semantics $\mathbf{Sem}^{mat}_{brave}$; it can be viewed as combining the idea of *FastEvol*[CKNZ10] with $\mathbf{Sem}^{mat}_2$ to handle inconsistencies by giving priority to triples that ought to be inserted, and deleting all those triples from the store that clash with the new ones.

Example 57 Ex. 56 in Sec. 5.2 provided a safe rewriting safe(u) of the update u from Ex. 54. According to Alg. 5.3, this safe update is rewritten to:

Algorithm 5.3: *Brave semantics* $\mathbf{Sem}^{mat}_{brave}$

Input: Materialised triple store $G = \mathcal{T}_G \cup \mathcal{A}_G$, SPARQL update $u(P_d, P_i, P_w)$
Output: $G^{\mathbf{Sem}^{mat}_{brave}}_{u(P_d,P_i,P_w)}$

1 $P'_d := P_d^{\text{caus}}$;
2 **foreach** *triple pattern* $(?X \text{ a } C)$ *in* P_i^{eff} **do**
3 **foreach** C' *s.t.* $C \sqsubseteq \neg C' \in \mathcal{T}_G$ *or* $C' \sqsubseteq \neg C \in \mathcal{T}_G$ **do**
4 **if** $(?X \text{ a } C') \notin P'_d$ **then**
5 $P'_d := P'_d \,.\, \{?X \text{ a } C'\}^{\text{caus}}$
6 **end**
7 **end**
8 **end**
9 **return** $G_{u(P'_d, P_i^{\text{eff}}, \{P_w\} P_d^{fvars})}$

```
DELETE {?X a :Manager . ?X1 :belongsTo ?X .
        ?Y a :Employee . ?Y :belongsTo ?Y1}
INSERT {?X :belongsTo ?Y . ?X a :Employee . ?Y a :Manager}
WHERE {{?X :worksFor ?Y
MINUS{{?X2 :worksFor ?X} UNION {?Y :worksFor ?Y2}}}
OPTIONAL {?X1 :worksFor ?X} OPTIONAL {?Y :worksFor ?Y1} }
```

The DELETE clause removes potential clashes for the inserted triples. Note that also property assertions implying clashes need to be deleted, which introduces fresh variables $?X1$ and $?Y1$. These variables have to be bound in the WHERE clause, and therefore P_d^{fvars} adds two optional clauses to the WHERE clause, which is a computationally reasonable implementation of the concept P^{fvars} from Def. 45. ■

The DELETE clause P'_d of the rewritten update is initialized in Alg. 5.3 with the set P_d of triples from the input update. Rewriting ensures that also all "causes" of deleted facts are removed from the store, since otherwise the materialisation will re-insert deleted triples. To this end, line 1 of Alg. 5.3 adds to P'_d all facts from which P_d can be derived. Then, for each triple implied by P_i (that is, for each triple in P_i^{eff}) the algorithm computes the patterns of clashing triples and adds them to the DELETE clause P'_d, along with their causes. Note that it suffices to only consider disjointness assertions that are syntactically contained in $\mathcal{T}_G$ (and not those implied by $\mathcal{T}_G$), since we assume that the store G is materialised. Finally, the WHERE clause of the rewritten update is extended to satisfy the syntactic restriction that all variables in P'_d must be bound: bindings of "fresh" variables introduced to P'_d due to the domain or range axioms in $\mathcal{T}_G$ are provided by the part P_d^{fvars}, cf. Def. 45 and Ex. 57. The rewritten update is evaluated over the triple store, computing its new materialised and consistent state.

In the RDFS$_\neg$ ontology language and under the restriction that only ABox updates are allowed, the brave semantics is a belief revision operator [Han99, Win05], performing a minimal change of the RDF graph (which due to materialisation can be seen both

Algorithm 5.4: *Cautious semantics* $\mathbf{Sem}^{mat}_{caut}$

Input: Materialised triple store $G = \mathcal{T}_G \cup \mathcal{A}_G$, SPARQL update $u(P_d, P_i, P_w)$
Output: $G^{\mathbf{Sem}^{mat}_{caut}}_{u(P_d,P_i,P_w)}$

1 $W := \{\,\mathsf{FILTER}(\mathit{False})\}$ // *neutral element w.r.t. union*
2 **foreach** $(?X\ \mathsf{a}\ C) \in P_i^{\text{eff}}$ **do**
3 | **foreach** C' *s.t.* $C \sqsubseteq \neg C' \in \mathcal{T}_G$ *or* $C' \sqsubseteq \neg C \in \mathcal{T}_G$ **do**
4 | | $\Theta^-_{C'} := \{\,\mathsf{FILTER}(\mathit{False})\}$
5 | | **foreach** $(?Y\ \mathsf{a}\ C') \in P_d^{\text{caus}}$ **do**
6 | | | $\Theta^-_{C'} := \Theta^-_{C'}\ \mathsf{UNION}\ \{P_w\theta[?Y \mapsto ?X]\}$
7 | | **end**
8 | | $W := W\ \mathsf{UNION}\ \{\{?X\ \mathsf{a}\ C'\}\ \mathsf{MINUS}\ \{\Theta^-_{C'}\}\}$
9 | **end**
10 **end**
11 $Q := \mathsf{ASK\,WHERE}\ \{\{P_w\}.\{W\}\};$
12 **if** $Q(G)$ **then**
13 | **return** G
14 **end**
15 **else**
16 | **return** $G^{\mathbf{Sem}^{mat}_{brave}}_{u(P_d,P_i,P_w)}$
17 **end**

as a deductive closure of the formula representing the ABox as well as the minimal model of this formula). There is a unique way of resolving inconsistencies since the only deduction rule with more than one ABox assertion in the premise, is the clash due to class disjointness (Fig. 51): assuming intrinsic consistency, the choice of which class membership assertion to remove in order to avoid clash is univocal (new knowledge is always preferred).

Theorem 2 *Alg. 5.3, given a SPARQL update u and a consistent materialised triple store $G = \mathcal{T}_G \cup \mathcal{A}_G$, computes a new consistent and materialised state w.r.t.* brave semantics. *The rewriting in lines 1–6 takes time polynomial in the size of u and $\mathcal{T}_G$.*

5.3.2 Cautious Semantics

Unlike $\mathbf{Sem}^{mat}_{brave}$, its *cautious* version $\mathbf{Sem}^{mat}_{caut}$ always gives priority to triples that are already present in the triple store, and dismisses any inserts that are inconsistent with it. We implement this semantics as follows: (i) the DELETE command does not generate inconsistencies and thus is assumed to be always possible; (ii) the update is actually executed only if the triples introduced by the INSERT clause do not clash with state of the triple graph *after all deletions have been applied.*

Cautious semantics thus treats insertions and deletions asymmetrically: the former depend on the latter but not the other way round. The rationale is that deletions never cause inconsistencies and can remove clashes between the old and the new data.

As in the case of brave semantics, cautious semantics is implemented using rewriting, presented in Alg. 5.4. First, the algorithm issues an ASK query to check that no clashes will be generated by the INSERT clause, provided that the DELETE part of the update is executed. If no clashes are expected, in which case the ASK query returns *False*, the brave update from the previous section is applied.

For a safe update $u = (P_d, P_i, P_w)$, the ASK query is generated as follows. For each triple pattern $\{?X \ \mathsf{a} \ C\}$ among the effects of P_i, at line 3 Alg. 5.4 enumerates all concepts C' that are explicitly mentioned as disjoint with C in $\mathcal{T}_G$. As in the case of brave semantics, this syntactic check is sufficient due to the assumption that the update is applied to a materialised store; by the same reason also no property assertions need to be taken into account.

For each concept C' disjoint with C, we need to check that a triple matching the pattern $\{?X \ \mathsf{a} \ C'\}$ is in the store G and will not be deleted by u. Deletion happens if there is a pattern $\{?Y \ \mathsf{a} \ C'\} \in P_d^{\text{caus}}$ such that the variable $?Y$ can be bound to the same value as $?X$ in the WHERE clause P_w. Line 6 of Alg. 5.4 produces such a check, using a copy of P_w, in which the variable $?Y$ is replaced by $?X$ and all other variables are replaced with distinct fresh ones. Since there can be several such triple patterns in P_d^{caus}, testing for clash elimination via the DELETE clause requires a disjunctive graph pattern $\Theta^-_{C'}$ constructed at line 6 and combined with $\{?X \, \mathsf{a} \, C'\}$ using MINUS at line 7.

Finally, the resulting pattern is appended to the list W of clash checks using UNION. As a result, $\{P_w\}.\{W\}$ queries for triples that are not deleted by u and clash with an instantiation of some class membership assertion $\{?X \ \mathsf{a} \ C\} \in P_i^{\text{eff}}$.

Theorem 3 *Alg. 5.4, given a SPARQL update u and a consistent materialised triple store $G = \mathcal{T}_G \cup \mathcal{A}_G$, computes a new consistent and materialised state w.r.t.* cautious semantics. *The rewriting in lines 1–8 takes time polynomial in the size of u and $\mathcal{T}_G$.*

Example 58 Alg. 5.4 rewrites the safe update safe(u) from Ex. 56 as follows:

```
ASK WHERE{{?X :worksFor ?Y
 MINUS{{?X1 :worksFor ?X} UNION {?Y :worksFor ?Y2}}}
 .{{?Y a :Employee} UNION {?X a :Manager}}}
```

Now, consider an update u' having both INSERT and DELETE clauses:

```
DELETE {?Y a :Manager} INSERT{?X a :Employee}
WHERE {?X :worksFor ?Y}
```

The update u' inserts a single class membership fact and thus is always intrinsically consistent[1]. The ASK query in Alg. 5.4 takes the DELETE clause of u' into account:

[1]That is, under the assumption that we do not have $A \ \mathsf{dw} \ A$ in TBox $\mathcal{T}_G$.

Algorithm 5.5: *Fainthearted semantics* $\mathbf{Sem}^{mat}_{faint}$

Input: Materialised triple store $G = \mathcal{T}_G \cup \mathcal{A}_G$, SPARQL update $u(P_d, P_i, P_w)$
Output: $G^{\mathbf{Sem}^{mat}_{faint}}_{u(P_d,P_i,P_w)}$

$W := P_w$
foreach *triple pattern* $(?X \text{ a } C)$ *in* P_i^{eff} **do**
  foreach C' *s.t.* $C \sqsubseteq \neg C' \in \mathcal{T}_G$ *or* $C' \sqsubseteq \neg C \in \mathcal{T}_G$ **do**
    $\Theta^-_{C'} := \{\,\mathsf{FILTER}(False)\}$;
    foreach $(?Z \text{ a } C') \in P_d^{\text{caus}}$ **do**
      $\Theta^-_{C'} := \Theta^-_{C'}$ UNION $\{P_w\theta[?Z \mapsto ?X]\}$;
    end
    $W := \{W\}\,\mathsf{MINUS}\,\{?X \text{ a } C'\,\mathsf{MINUS}\,\{\Theta^-_{C'}\}\}$;
  end
end
$W := \{W\}\,\mathsf{UNION}\,\{P_w\theta_1 \,.\, P_d^{fvars}\theta_1\}$;
return $G_{u(P_d^{\text{caus}}\theta_1,\ P_i^{\text{eff}},\ W)}$

```
ASK WHERE {{?X :worksFor ?Y}
.{{?X a :Manager} MINUS {?Z :worksFor ?X }}}
```

■

5.3.3 Fainthearted Semantics

Our third, *fainthearted* semantics is meant to take an intermediate position between the cautious semantics and the brave one. A shortcoming of the cautious semantics is that massive update can be retracted because of only a few clashing triples. Not to discard an update completely in such a case, the user can decide either to override the existing knowledge—that is, opt for the brave semantics—or to apply insertions only for those variable bindings which are not clashing with the existing state, which is what the fainthearted semantics does.

Our realization of the idea of accommodating non-clashing inserts is based on *decoupling the insert and the delete* part of an update: whereas the delete is executed for *all* variable bindings satisfying the WHERE clause, one dismisses the inserts for variable bindings that yield clashes with the state of the store *after the delete*. That is, we deviate from the notion of update as an atomic operation in a different way than in the safe rewriting where *both* deletions and insertions are dismissed for variable bindings leading to clashes. Our motivation for such a design decision is explained next.

Assume that for each variable binding μ returned by the WHERE pattern, we want to either insert $ground(P_i\mu)$—i.e., the set of ground triples in pattern P_i, see Def. 24—along with deleting $ground(P_d\mu)$, or dismiss μ altogether. As an example, consider the update u' from Ex. 58 and the ABox {`:john :worksFor :anna. :john a :Manager. :bob :worksFor :john`}. With the variable binding $\mu_1 = [?X \mapsto$

:john, $?Y \mapsto$:anna] we insert `:john a :Employee` knowing that the clashing fact `:john a :Manager` will be deleted by the binding $\mu_2 = [?X \mapsto$:bob, $?Y \mapsto$:john]. However, if the update is atomic, this anticipated deletion will only happen if $\mathrm{gr}(P_i\mu_2)$ does not introduce clashes. Assume this is the case (i.e. also `{:bob a :Manager}` is in the ABox): we have to look one more step ahead and check if this triple will be deleted by some variable binding μ_3, and so on. This behaviour could be realized with SPARQL path expressions, which would however stipulate severe syntactic restrictions on the WHERE clause P_w of the original update.

As mentioned above, our interpretation of fainthearted semantics assumes independence between the INSERT and DELETE parts of the update. To implement this, we rely on SPARQL's flexible handling of variable bindings. Namely, we rename the variables in the DELETE clause apart from the rest of the update, and put this renamed apart copy of the WHERE clause in a new UNION branch. The original WHERE clause is then rewritten (using MINUS operator, similarly to the case of cautious semantics) to ensure that insertions are only done for variable bindings where clashes are removed by the DELETE clause with some variable binding. The implementation can be found in Alg. 5.5.

Example 59 The update u' from Example 58 is rewritten as follows by Alg. 5.5:

```
DELETE {?Y1 a :Manager } INSERT {?X a :Employee}
WHERE {{?X2 :worksFor ?Y1} UNION {?X :worksFor ?Y.
      {MINUS {?X a :Manager MINUS {?X3 :worksFor ?X}}}}}
```

The first union branch binds the variables in the DELETE clause (both using fresh variables). The second branch binds the variable `?X` in the INSERT clause, using MINUS to remove variable bindings for which a non-deleted clash exists. The test that a clash will not be deleted is expressed using the inner MINUS operator. ■

We conclude with a claim of correctness and polynomial complexity of the rewriting, similar to those made for the brave and cautious semantics.

Theorem 4 *Alg. 5.5, given a SPARQL update u and a materialised triple store $G = \mathcal{T}_G \cup \mathcal{A}_G$ w.r.t.* fainthearted semantics*, computes a new consistent and materialised state. The rewriting in lines 1–9 takes time polynomial in the size of u and $\mathcal{T}_G$.*

5.4 Postulates for Mat-Preserving Semantics

In the previous sections, we have defined three materialised- and consistency-preserving semantics for SPARQL updates, namely $\mathbf{Sem}^{mat}_{brave}$, $\mathbf{Sem}^{mat}_{caut}$ and $\mathbf{Sem}^{mat}_{faint}$. In this section we consider the rationality of these update semantics by checking them against the postulates we defined in Sec. 3.2. A summary of the semantics and the postulates they satisfy is given in Table 52. One can see that $\mathbf{Sem}^{mat}_{brave}$ fulfills the most number of

	K1	K∗2	K–2	K∗3	K–3	K∗4	K–4	K∗5	K∗5′	K–5	K–5′	K–5″	K–5‴	K6	Total /14
$\mathbf{Sem}^{mat}_{brave}$	✓	✓	✓	✓	✓	✓	✓	(✓)	(✓)	X	X	X	X	X	9
$\mathbf{Sem}^{mat}_{caut}$	✓	X	✓	✓	✓	✓	✓	(✓)	(✓)	X	X	X	X	X	8
$\mathbf{Sem}^{mat}_{faint}$	✓	X	✓	✓	✓	✓	✓	(✓)	(✓)	X	X	X	X	X	8

Table 52: Checking postulates K1-K6 against $\mathbf{Sem}^{mat}_{brave}$, $\mathbf{Sem}^{mat}_{caut}$ and $\mathbf{Sem}^{mat}_{faint}$

postulates, precisely one postulate more than $\mathbf{Sem}^{mat}_{caut}$ and $\mathbf{Sem}^{mat}_{faint}$. In the following we give more explanation on the fulfillment of each semantics vs. postulates.

$\mathbf{Sem}^{mat}_{brave}$ satisfies K1 because it relies upon $\mathbf{Sem}^{mat}_{2}$, which in turn is a materialise-preserving semantics. It satisfies K∗2 because the semantics gives priority to triples ought to be inserted, consequently resulting with these triples being added in the triple store. It satisfies K–2 because the deletion of triples via the semantics of SPARQL/Update does not add new triples, and this holds for other semantics as well. K∗3 is satisfied because for $u = \texttt{INSERT}\{\mathcal{A}\}$ the semantics is subsumed by $\mathbf{Sem}^{mat}_{expand}$, given that u could potentially trigger additional deletions as defined in Alg. 5.3. It satisfies K–3 because the deletion of a designated triple that does not exist in the triple store, plus its causes results with no changes; the same holds for the other semantics. That is, if causes would exist in $mat(G)$, then also the designated triple would exist in $mat(G)$. K∗4 is satisfied because in the case where inserts cause no inconsistency, then the semantics coincides with $\mathbf{Sem}^{mat}_{expand}$; the same holds for the other semantics. It satisfies K–4 because according to the definition, triples plus their causes are removed from the materialised triple store. K∗5 holds because we have two options: (i) we refuse intrinsic inconsistencies in which case it holds, or (ii) we apply safe rewriting which tries to find a maximal consistent subset of the update and therefore could be interpreted as a kind of para-consistent approach. We thus denote it in brackets (✓), and this likewise holds for the other semantics. Its variant K∗5′ also holds in the same way because we apply the semantics to resolve inconsistencies; likewise holds for the other semantics. K–5, K–5′, K–5″, K–5‴ are not satisfied because insertion might result with triples ought to be inserted plus effects, whereas deletion might result with triples ought to be deleted plus the causes, where the causes are disjoint from effects, see Ex. 30 in the context of $\mathbf{Sem}^{mat}_{2}$. It does not satisfy K6 because given $u_1 = \texttt{DELETE}\{\mathcal{A}_1\}$, $u_2 = \texttt{DELETE}\{\mathcal{A}_2\}$ such that $\mathcal{A}_1 \subset \mathcal{A}_2$ and $mat(\mathcal{T}_G \cup \mathcal{A}_1) = mat(\mathcal{T}_G \cup \mathcal{A}_2)$, then $G^{sem}_{u_1} \neq G^{sem}_{u_2}$; same holds for inserts, as well as for the other semantics.

$\mathbf{Sem}^{mat}_{caut}$ satisfies K1 because it uses $\mathbf{Sem}^{mat}_{brave}$, and thus relies upon $\mathbf{Sem}^{mat}_{2}$ which is materialise-preserving semantics. It does not satisfy K∗2 because it gives no priority to triples ought to be inserted. It satisfies K–2 as with $\mathbf{Sem}^{mat}_{brave}$. K∗3 is satisfied because for $u = \texttt{INSERT}\{\mathcal{A}\}$ the semantics is subsumed by $\mathbf{Sem}^{mat}_{expand}$, as it relies upon $\mathbf{Sem}^{mat}_{brave}$. Likewise, as with $\mathbf{Sem}^{mat}_{brave}$, it satisfies K–3, K∗4, K–4, K∗5, K∗5′ and likewise, it does not satisfy K–5, K–5′, K–5″, K–5‴ and K6.

$\mathbf{Sem}^{mat}_{faint}$ is same as $\mathbf{Sem}^{mat}_{caut}$.

In the next chapter we are going to investigate updating Wikipedia via DBpedia mappings and SPARQL. In this case not only the ontology is more expressive than RDFS$_\neg$ but also the mappings are present. This makes the problem even more challenging, for which user input is very important and crucial in resolving (disambiguating) updates.

CHAPTER 6

Updating Wikipedia via DBpedia Mappings and SPARQL

In the previous Chapter 5 we discussed updates over RDFS$_\neg$ ontologies, and how different semantics inspired by belief revision can be adopted and implemented using SPARQL rewritings in order to deal with inconsistencies. In this chapter, we are going to see how we can re-use these update semantics and lift them in more expressive ontology language (DBpedia ontology) and physical (DBpedia) mappings, concretely, in the case of updating Wikipedia using DBpedia mappings and SPARQL. Referring to Fig. 11 the focus is in the update semantics that are materialise preserving in the context of ABox updates for the DBpedia OWL fragment.

In this chapter, we present an approach to allow ontology-based updates of wiki content. Starting from DBpedia-like mappings converting infoboxes to a fragment of `OWL 2 RL` ontology, we discuss various issues associated with translating SPARQL updates on top of semantic data to the underlying Wiki content. On the one hand, we provide a formalization of DBpedia as an Ontology-Based Data Management framework. On the other hand, we provide a novel approach to the inherently intractable update translation problem, leveraging the pre-existent data for disambiguating updates.

DBpedia [LIJ+15] is a community effort that has created the most important cross-domain dataset in RDF [BGe04] in the focal point of the Linked Open Data (LOD) cloud [ABK+07]. At its core is a set of declarative mappings extracting data from Wikipedia *infoboxes* and tables into RDF. However, DBpedia makes knowledge machine readable only, rather than also *machine writable*. This not only restricts the possibilities of automatic curation of the DBpedia data that could be semi-automatically propagated back to Wikipedia, but also prevents maintainers from evaluating the impact of their edits on the consistency of knowledge (across infoboxes); indeed, previous work confirms that there are such inconsistencies discoverable in DBpedia [BKPR14, DKF+15] arising

most likely from inconsistent content in Wikipedia itself with respect to the mappings and the DBpedia ontology. Excluding the DBpedia taxonomy from the editing cycle is thus a—as we will show, unnecessary—drawback, but rather can be turned into an advantage for helping editors to create and maintain consistent content inside infoboxes, which we aim to address.

To this end, in this chapter we want to make a case for DBpedia as a practical, real-world benchmark for Ontology-Based Data Management (OBDM) [Len18]. Although based on fairly restricted mappings—which we cast as a variant of so-called nested tuple-generating dependencies (tgds) herein—and minimalistic TBox language, accommodating DBpedia updates is intricate from different perspectives. The challenges are both:

- conceptual: What is an adequate semantics for DBpedia SPARQL updates? and
- practical: How to cope with high ambiguity of update resolutions?

While general updates in OBDM remain largely infeasible, we still arrive at reasons to believe, that for certain use cases within DBpedia updates, reasonable and practically usable conflict resolution policies could be defined; we present the first serious attempt with DBpedia as a potential benchmark use case in this area.

Pushing towards the vision of a "Read/Write" Semantic Web, the unifying capabilities of SPARQL extend beyond the mere querying of heterogeneous data. Indeed, the standardization of update functionality introduced in SPARQL 1.1 renders SPARQL as a strong candidate for the role of web data manipulation language. For a concrete motivation example consider Listing 1, where a simple SPARQL Update request would reflect a recent merger of French administrative regions: for each settlement belonging to either Upper or Lower Normandy, we set the corresponding administrative attribution property to be just Normandy. In our scenario, the user should have means to write this update in SPARQL and let it be reflected in the underlying Wikipedia data.

Despite clear motivation, updates in the information integration setting abound with all sorts of challenges, starting from obvious data security concerns, to performance, data quality issues and, last but not least the technical issues of side effects and lack of unique semantics, demonstrated already in the classical scenarios such as database views and deductive databases [BS81, CFG$^+$12]. Although based on a very special join-free mapping language, the DBpedia setting is not different in this respect. With a high-quality curated data source at the backend, we set our goal not at ultimate transparency and automatic translation of updates, but rather at maximally support users in choosing the most economic and *typical* way of accommodating an update while maintaining (or at least, not degrading) consistency and not losing information inadvertently. As for DBpedia, if such RDF frontend systems have their own taxonomy (TBox) with also class and property disjointness assertions as well as functionality of properties, updates can result in inconsistencies with the data already present. In particular, in the next sections our focus will be on:

```
DELETE { ?X :region :Upper_Normandy . ?Y :region :Lower_Normandy .}
INSERT { ?X :region :Normandy . ?Y :region :Normandy}
WHERE { {?X :region :Upper_Normandy} UNION {?Y :region :Lower_Normandy} }
```

Listing 1: SPARQL 1.1. Update that merges two regions in France.

- we formalize the actual ontology language used by DBpedia as an `OWL 2 RL` fragment, and DBpedia mappings as a variant of so-called nested tuple-generating dependencies (tgds); based on this formalization
- we propose a semantics of OBDM updates for DBpedia and its Wikipedia mappings
- we discuss how such updates can be practically accommodated by suitable conflict resolution policies: the number of consistent revisions are in the worst case exponential in the size of the mappings and the TBox, so we investigate policies for choosing the "most reasonable" ones, e.g. following existing patterns in the data, that is choosing most popular fields in the present data to be filled upon inserts.

Note that, since neither the SPARQL Update language [GPP13] nor the SPARQL Entailment regimes [GOH+13] specification covers the behaviour of updates in the presence of TBox axioms, the choice of semantics in such cases remains up to application designers. In Chapter 4 and Chapter 5 we have discussed how SPARQL updates relating to the ABox can be implemented with TBoxes allowing no or limited form of inconsistency (class disjointness), a work we partially build upon herein: as a requirement from this prior work (as a consequence of the common postulates for updates in belief revision), such an update semantics needs to ensure that no mutually inconsistent pairs of triples are inserted in the ABox. In order to achieve this, a policy of conflict resolution between the new and the old knowledge is needed.

To this end, in Chapter 5 we defined *brave*, *cautious* and *fainthearted* semantics of updates. Brave semantics removes from the knowledge base all facts clashing with the inserted data. Cautious semantics discards entirely an update if it is inconsistent w.r.t. knowledge base, otherwise brave semantics is applied. Fainthearted semantics is in-between the two, amounts to adding an additional filter to the WHERE clause of SPARQL update in order to discard variable bindings which make inserted facts contradict prior knowledge. In this chapter, we stick to these three basic cases, extending them to the OWL fragment used by DBpedia. However, since our goal is to accommodate updates as Wiki infobox revisions for which no batch update language exists, we restrict our considerations to *ground updates* (u^+, u^-) of triples over URIs and literals that are to be inserted or, respectively, deleted (instead of considering the whole general SPARQL Update language).[1]

We now formalise the DBpedia setting in the context of OBDM.

[1] We emphasize though that such an extension is a fairly straightforward extension of the discussions in Chapter 5, since general SPARQL Updates can be viewed as templates which are instantiated into exactly such sets of INSERTed and DELETed triples.

6.1 (OWL) DBpedia Setting

We define the declarative **WikiDBpedia framework** (WDF) $\mathcal{F}$ as a triple $(\mathbf{W}, \mathcal{M}, \mathcal{T})$ where $\mathbf{W}$ is a relational schema equipped with integrity constraints (described later) encoding the infoboxes, $\mathcal{M}$ is a set of rules transforming it into RDF triples (the DBpedia ABox), and $\mathcal{T}$ is a TBox. The rules in $\mathcal{M}$ are given by a custom-designed declarative DBpedia mapping language [JSIB10]. This language can be captured by the language of *nested tuple generating dependences* (nested tgds) [FHH+06, KPSS14], enhanced with negation in the rule bodies and interpreted functions for arithmetics, date, string and geocoordinate processing.

A *WDF instance* of a WDF $(\mathbf{W}, \mathcal{M}, \mathcal{T})$ is an infobox instance I satisfying $\mathbf{W}$. We now specify the language used to formalize the TBox $\mathcal{T}$, the tgds language of $\mathcal{M}$ and the infobox schema $\mathbf{W}$.

DBpedia ontology language (DBP). DBpedia uses a fragment of OWL 2 RL profile, which we call DBP. It includes the RDF keywords `subClassOf` (which we abbreviate as sc), `subPropertyOf` (sp), `domain` (dom) and `range` (rng), `disjointWith` (dw), `propertyDisjointWith` (pdw), `inversePropertyOf` (inv), `differentFrom` (df) as well as `functionalProperty` (func). At present, functional properties in DBpedia are limited to data properties, and inverse functional roles are not used. Inference rules for the ontology language DBP are summarized in Figure 61.

Many concepts in the actual DBpedia are copied from external ontologies like Yago [SKW07] and UMBEL[2]. All DBpedia resources also instantiate the concepts in DBpedia ontology, with the namespace `http://dbpedia.org/ontology`. They can be listed by the following SPARQL query:

```
SELECT DISTINCT ?x WHERE {{?x a owl:Class}
   UNION {?x a owl:ObjectProperty}
   UNION {?x a owl:DatatypeProperty}
   FILTER(strstarts(str(?x), "http://dbpedia.org/"))}
```

As of October 2017, this query retrieves 761 concepts, 1125 object and 1780 datatype properties for the English Live DBpedia[3]. Herewith, we only consider the facts from

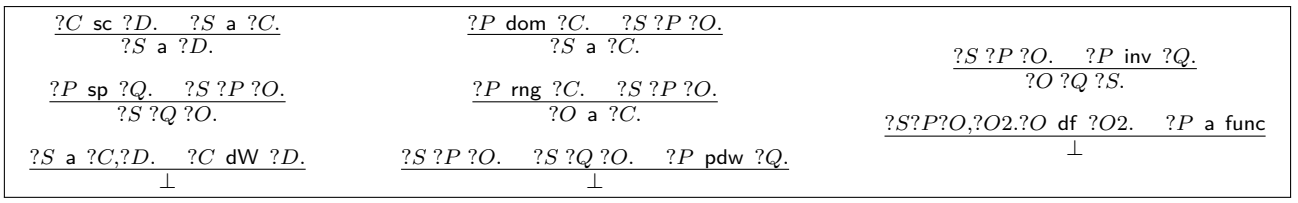


Figure 61: DBpedia RDFS plus OWL rules comprising a fragment of `OWL 2 RL`.

[2] `http://techwiki.umbel.org/index.php/UMBEL_Vocabulary`
[3] `http://live.dbpedia.org/sparql`

Type of Mappings	Declared	Description
Template	958	Map Wiki (infobox) templates to DBpedia classes.
Property	19,972	Map Wiki (infobox) template properties to DBpedia properties.
IntermediateNode	107	Generate a blank node with a URI.
Conditional	31	Depend on (infobox) template properties and their values.
Calculate	23	Compute a function over two properties.
Date	106	Mappings that generate a starting and ending date.

Table 61: Description of DBpedia (English) mappings.

this core vocabulary set instantiated with the set of DBpedia mappings $\mathcal{M}$, and not the imported assertions from the external ontologies. We denote this vocabulary by $\mathbf{T}$ and, analogously to the infobox part of the system, call it "schema".

Infobox schema W. Each Wiki page is identified by a URI which translates to a subject IRI in DBpedia. We model this semistructured data store using a relational schema $\mathbf{W}$ with two ternary relations $W_i = \mathsf{UTI}$ and $W_d = \mathsf{IPV}$, attribute I storing infobox identifiers, U page URI, T infobox type, and P and V being respectively property names and values. That is, unlike the real Wiki where infoboxes may belong to different pages or be separate tables of distinct types, we use an auxiliary surrogate key I to horizontally partition the single key-value store W_d. Our schema $\mathbf{W}$ assumes key constraints $\mathsf{UT} \to \mathsf{I}$, $\mathsf{IP} \to \mathsf{V}$ and the inclusion dependency $W_d[\mathsf{I}] \subseteq W_i[\mathsf{I}]$ which we encode as the set of rules $\mathcal{W}$:

$$\begin{aligned}\mathcal{W} = \{&\forall i \forall p \forall v (W_d(i,p,v) \to \exists u \exists t\ W_i(u,t,i)),\\ &\forall u \forall t \forall i_1 \forall i_2 (W_i(u,t,i_1) \wedge W_i(u,t,i_2) \wedge i_1 \neq i_2 \to \bot),\\ &\forall i \forall p \forall v_1 \forall v_2 (W_d(i,p,v_1) \wedge W_d(i,p,v_2) \wedge v_1 \neq v_2 \to \bot)\}.\end{aligned}$$

Mapping constraints $\mathcal{M}$. The specification [JSIB10] distinguishes several types of DBpedia mappings summarized in Table 61 along with their figures in the English DBpedia. All these mappings can be represented as *nested tgds* [FHH$^+$06, KPSS14] extended with *negation and constraints* in the antecedents for capturing the conditional mappings and interpreted functions in the conclusions of implications, in the case of calculated mappings handling, e.g., dates or geo-coordinates. A crucial limitation of the mapping language (which we call *DBpedia tgds*) is the *impossibility of comparisons between infobox property values*. Infobox type $W_i.\mathsf{T}$ and property names $W_d.\mathsf{P}$ must be specified explicitly.

Definition 48 (Nested tgds [KPSS14]) *Fix a partition of the set of first-order variables into two disjoint infinite sets, X and Y. A* nested tgd *is a first-order formula that can be generated by the following recursive definition:*

$$\chi ::= \alpha \mid \forall \vec{x}\ (\beta_1 \wedge \ldots \wedge \beta_k \to \exists \vec{y}\ (\chi_1 \wedge \ldots \wedge \chi_l))$$

where each $x_i \in X$, each $y_i \in Y$, α is an atomic formula over the target schema, and each β_j is an atomic formula over the source schema containing only variables from X,

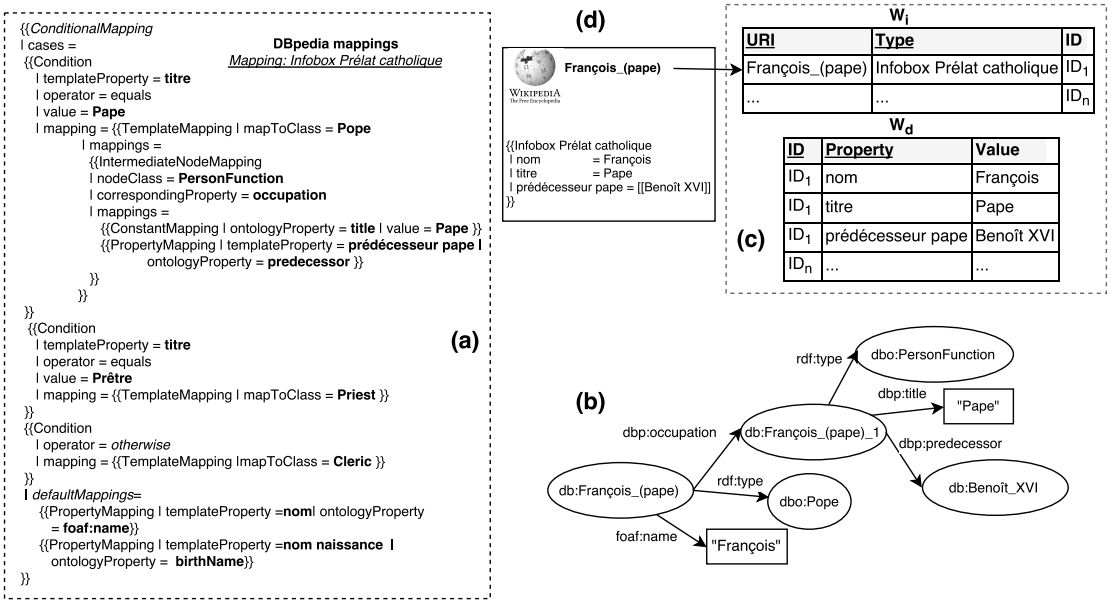


Figure 62: (a) DBpedia mappings, (b) the RDF graph, and the Infobox as an instance of the schema **W** (c) and in the native format (d).

such that each x_i occurs in some β_j. In our case, the source schema is an infobox schema, and the target schema is an ontology.

Example 60 The following formula is an example of a nested tgd ([KPSS14]):

$$\forall x_1 \forall x_2 (P(x_1, x_2) \rightarrow \exists y (P'(y, x_2) \wedge \forall x_3 (P(x_1, x_3) \rightarrow P'(y, x_3))))$$

■

Example 61 Fig. 62(a) shows a conditional mapping transferring the information about clerics from French wiki pages with an infobox *Prélat catholique* (d). Under these conditions, the excerpt shown in Fig. 62(c) as an instance over the schema **W** gives rise to the triples depicted in Fig. 62(b). A tgd formalizing a French DBpedia mapping for clergy is given below:

$$\begin{aligned}
&\forall U \forall I \big(W_i(U, \text{'fr:Prélat catholique'}, I) \rightarrow \\
&\quad \big(W_d(I, \text{'titre'}, \text{'Pape'}) \rightarrow \\
&\qquad \exists Y \big(\mathsf{Pope}(U) \wedge \mathsf{occupation}(U, Y) \wedge \mathsf{PersonFunction}(Y) \\
&\qquad\quad \wedge \mathsf{title}(Y, \text{'Pape'})) \quad \textit{// ``Intermediate node mapping''} \\
&\qquad\quad \wedge \ldots \\
&\qquad\quad \wedge \forall X (W_d(I, \text{'prédécesseur pape'}, X) \rightarrow \mathsf{predecessor}(Y, X)) \big) \big)
\end{aligned}$$

$$
\begin{aligned}
&\ldots \\
&\wedge\ (W_d(I, \text{'titre'}, \text{'Prêtre'}) \rightarrow \mathsf{Priest}(U)) \\
&\quad \textit{// The ``otherwise'' branch:} \\
&\wedge\ (\neg W_d(I, \text{'titre'}, \text{'Pape'}) \wedge \ldots \wedge \neg W_d(I, \text{'titre'}, \text{'Prêtre'}) \rightarrow \mathsf{Cleric}(U)) \\
&\wedge\ \forall X(W_d(I, \text{'nom'}, X) \rightarrow \mathsf{foaf{:}name}(U, X)) \\
&\ldots \\
&\wedge\ \forall X(W_d(I, \text{'nom naissance'}, X) \rightarrow \mathsf{birthName}(U, X)))\big)
\end{aligned}
$$

The specification stipulates that conditions are evaluated in the natural order, and thus every next condition has to include the negation of all preceding conditions. In our case, this is only illustrated by the last, default ("otherwise") case, since the conditions are mutually exclusive. Note also that no universally quantified variable besides the page URI U and the technical infobox identifier I)—i.e., no X variable representing an infobox property—can occur more than once on the left-hand side of an implication, due to the lack of support for comparisons between infobox properties. ■

One further particularity of the chase with tgds is the handling of existentially quantified variables that represent so-called "intermediate nodes" (e.g., Y in Ex. 61). A usual approach is to instantiate such variables by null values, which could become blank nodes on the RDF storage side. The strategy currently followed by DBpedia is different: instead of blank nodes, the chase produces fresh IRIs. By appending an incremented number to the Wiki page address it avoids clashes with existing page URIs. We name it *constant inventing chase*.

Updates. We consider updates that can be specified on both the infobox and the DBpedia sides. Since DBpedia is a materialised extension constructed based on the contents of infoboxes, persistent modifications must be represented as infobox updates. We consider updates based on ground facts to be inserted or deleted, each update being limited to exactly one schema, the infobox **W** or DBpedia **T**.

Definition 49 *Let* **S** *be a schema and J an instance of* **S**. *An* update *u of J is a pair (u^-, u^+) of sets of ground atoms over* ***S*** *in which u^+ signifies facts to be inserted to J and u^- facts to be removed from J. Deletions are applied prior to insertions.*

Since WDF includes the mapping and TBox rules, special care is needed to make update effective and enforce or maintain the consistency of the affected WDF instance apply a minimal necessary modifications. Our formalization is close to the usual definition of formula based belief revision operators. A WDF instance I is identified with a conjunctive formula over **W** that satisfies integrity constraints $\mathcal{W}$ of the infobox schema. The notation $u(I)$ is understood as $(I \setminus u^-) \cup u^+$ where $I \setminus u^-$ denotes the removal of all conjuncts occurring in u^- from I, and $I \cup u^+$ is the same as the conjunction $I \wedge u^+$.

We define a partial order $\preceq$ relation between updates as follows $u \preceq e$ iff $u^- \subseteq e^-$ and $u^+ \subseteq e^+$. One can as well consider other, e.g. cardinality based, partial orders.

Definition 50 *Let $\mathcal{F}$ be a WDF $(\mathbf{W}, \mathcal{M}, \mathcal{T})$, such that $\mathbf{W}$ is equipped with the integrity constraints $\mathcal{W}$, I be an $\mathcal{F}$-instance and let u be an update over $\mathcal{T}$. The consistency-oblivious semantics $\{\![u^{\mathbf{W}}]\!\}$ of u is the set of smallest (w.r.t. $\preceq$) updates $[u^{\mathbf{W}}]$ over the infobox schema $\mathbf{W}$ such that the conditions $[u^{\mathbf{W}}](I) \cup \mathcal{W} \cup \mathcal{M} \cup \mathcal{T} \not\models u^-$, $[u^{\mathbf{W}}](I) \cup \mathcal{W} \cup \mathcal{M} \cup \mathcal{T} \models u^+$ and $I \cup \mathcal{W} \not\models \bot$ hold.*

The former two conditions ensure the effectiveness of the update, that is, that all desired insertions and deletions are performed. The conformance with $\mathbf{W}$ ensures that the update can be accommodated in the physical infobox storage model, which the constraints $\mathcal{W}$ simulate. The following definition of the semantics $\{\![\![u^{\mathbf{W}}]\!]\!\}$ restricts the semantics $\{\![u^{\mathbf{W}}]\!\}$ in order to ensure that the DBpedia instance can be used under entailment w.r.t. $\mathcal{T}$, denoted as closure $cl(I, \mathcal{M})$. Note that both semantics $\{\![u^{\mathbf{W}}]\!\}$, $\{\![\![u^{\mathbf{W}}]\!]\!\}$ depend on $\preceq$, $\mathcal{F}$ and on I—which is not explicit in our notation for the sake of readability.

Definition 51 *Let $\mathcal{F}$ be a WDF $(\mathbf{W}, \mathcal{M}, \mathcal{T})$, such that $\mathbf{W}$ is equipped with the integrity constraints $\mathcal{W}$, I be an $\mathcal{F}$-instance and let u be an update over $\mathcal{T}$. The consistency-aware semantics $\{\![\![u^{\mathbf{W}}]\!]\!\}$ of u is the set of smallest (w.r.t. $\preceq$) updates $[\![u^{\mathbf{W}}]\!]$ such that $[\![u^{\mathbf{W}}]\!] \in \{\![u^{\mathbf{W}}]\!\}$ and $[\![u^{\mathbf{W}}]\!](I) \cup \mathcal{W} \cup \mathcal{M} \cup \mathcal{T} \not\models \bot$.*

6.2 Challenges of DBpedia OBDM

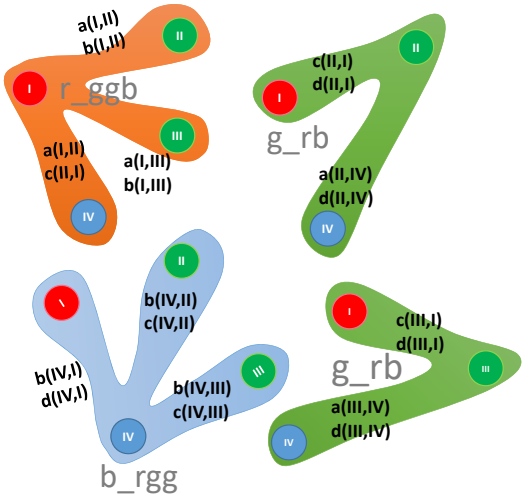
We consider the Existence of solutions problem and show that it is in general intractable even for the consistency-oblivious semantics.

Problem ExSol-Obl. Parameter: WDF $\mathcal{F} = (\mathbf{W}, \mathcal{M}, \mathcal{T})$. Input: $\mathcal{F}$-instance I, update u. Test if $\{\![u^{\mathbf{W}}]\!\} \neq \emptyset$.

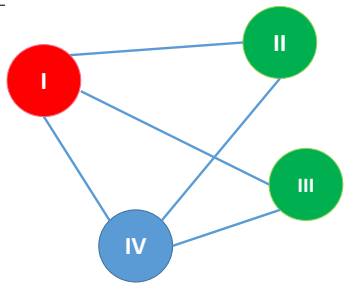
Proposition 8 ExSol-Obl *is NP-complete.*

Proof 2 (Sketch) Consider a DBpedia update u, and the WDF instance I. For the membership in NP, observe that enforcing the constraints in $\mathcal{M}$ and in $\mathcal{T}$ (e.g., via chase) terminates in polynomial time for every fixed WDF $\mathcal{F}$, which gives a bound on the size of the infobox instance witnessing $\{\![u^{\mathbf{W}}]\!\} \neq \emptyset$ for an instance I. For each condition in the mapping $\mathcal{M}$ (limited to comparing a single infobox value with a fixed constant), we can define a canonical way of satisfying it, and thus defining *canonical witnesses*, whose size and active domain is determined by u, I and $\mathcal{F}$. As a result, the test comes down to guessing a canonical witness and checking it by the chase with constraints, that u^+ is inserted and u^- deleted, which is feasible in poly time for the constraints in DBP.

For the hardness, consider the following reduction from the 3-Colorability problem. Let I be empty and let the set of atoms A that the DBpedia update $u = (\emptyset, A)$ inserts represents an undirected graph $G = (V, E)$ of degree at most 4 (for which 3Col is intractable [GJS76]). A represents the vertices V as IRIs and each edge $(x, y) \in E$ for



(a) The 1-neighborhoods of G



(b) The graph G

	red	*green*	*blue*
red		*ab*	*ac*
green	*cd*		*ad*
blue	*bd*	*bc*	

(c) Rules for TBox property generation

Figure 63: Concepts of the proof of Prop. 8

the IRIs x, y is represented by a collection of 8 atoms of the form $\mathsf{a}(x,y)$, $\mathsf{a}(y,x)$, $\mathsf{b}(x,y)$, $\mathsf{b}(y,x)$, $\mathsf{c}(x,y)$, $\mathsf{c}(y,x)$, $\mathsf{d}(x,y)$, and $\mathsf{d}(y,x)$, for which the assertions $\mathsf{a} = \mathsf{a}^{-1}$, $\mathsf{b} = \mathsf{b}^{-1}$, $\mathsf{c} = \mathsf{c}^{-1}$ and $\mathsf{d} = \mathsf{d}^{-1}$ are defined in $\mathcal{T}$.

Each infobox encodes a single vertex of the graph, together with all its adjacent vertices (at most four direct neighbors). Together these 1-neighborhoods cover the graph. The encoding ensures that the only way to obtain the regular DBpedia representation of the graph, with exactly eight property assertions for each pair of vertices, is only possible if every vertex is assigned the same color in each infobox. This is achieved by distributing the a, b, c and d between each pair of adjacent nodes depending on the node color. The rules for that are given in Fig. 2(c). For instance, an edge between a red I and a green vertex II is composed from the properties $\mathbf{a}$(I,II) $\mathbf{b}$(I,II) whose creation is triggered by the infobox of page I, and the other two properties b, c are created by chasing the infobox I: $\mathbf{c}$(II,I), $\mathbf{d}$(II,I). Due to symmetry, this results in the eight property assertions.

The excerpt of the mapping for the neighborhood types 'r_ggb', 'b_rgg', 'g_rb' illustrated by a graph in Figure 2 (b) is shown below.

$$
\begin{aligned}
&\forall U \forall I \big(W_i(U, \text{'vertex'}, I) \rightarrow \\
&\quad \big(W_d(I, \text{'n-type'}, \text{'r\_ggb'}) \rightarrow \big(\mathsf{Node}(U) \wedge \forall X \, (W_d(I, \text{'n1'}, X) \rightarrow \mathsf{a}(U,X) \wedge \mathsf{b}(U,X)) \\
&\qquad \wedge \forall X \, (W_d(I, \text{'n2'}, X) \rightarrow \mathsf{a}(U,X) \wedge \mathsf{b}(U,X)) \wedge \forall X \, (W_d(I, \text{'n3'}, X) \rightarrow \mathsf{a}(U,X) \wedge \mathsf{c}(U,X)) \big) \\
&\wedge \big(W_d(I, \text{'n-type'}, \text{'b\_rgg'}) \rightarrow \big(\mathsf{Node}(U) \wedge \forall X \, (W_d(I, \text{'n1'}, X) \rightarrow \mathsf{b}(U,X) \wedge \mathsf{d}(U,X)) \\
&\qquad \wedge \forall X \, (W_d(I, \text{'n2'}, X) \rightarrow \mathsf{b}(U,X) \wedge \mathsf{b}(U,X)) \wedge \forall X \, (W_d(I, \text{'n3'}, X) \rightarrow \mathsf{b}(U,X) \wedge \mathsf{c}(U,X)) \big) \\
&\wedge \big(W_d(I, \text{'n-type'}, \text{'g\_rb'}) \rightarrow \big(\mathsf{Node}(U) \wedge \forall X \, (W_d(I, \text{'n1'}, X) \rightarrow \mathsf{c}(U,X) \wedge \mathsf{d}(U,X)) \\
&\qquad \wedge \forall X \, (W_d(I, \text{'n2'}, X) \rightarrow \mathsf{a}(U,X) \wedge \mathsf{d}(U,X)) \big) \\
&\wedge \ldots \text{etc for other 1-neighborhood types} \ldots) \blacksquare
\end{aligned}
$$

In the above proof, the whole graph is encoded in the update, which limits the severity of the challenge: one can argue that this is irrelevant if updates are small (and this is actually the case in our pragmatic approach). The construction can be modified to illustrate the complexity of the OBDM problem even for updates of fixed size. Consider a DBpedia instance representing a graph, and an DBpedia update u adding a single vertex (with up to four edges) to it. The extended graph must not be 3-colorable even if the original graph was, and thus checking $u \in \{\![u^{\mathbf{W}}]\!\}$, for a fixed size u, takes checking the 3 colorability of an extended graph of size $n+1$, which is NP-complete in n. We thus obtain

Proposition 9 *The* Decision OBDM problem *under the consistency-oblivious semantics is NP-complete even for updates of fixed size.*

If we bring the TBox and infobox schema constraints along with non-monotonicity of mapping rules into the picture, the potential challenges of accommodating updates start piling up quickly. An interplay of the following features of the framework can make update translation unwieldy: (i) inconsistencies due to the TBox assertions, namely the class and role disjointness and functional properties; (ii) many-to-many relationships between infobox and ontology properties defined by the mappings, and (iii) infobox schema constraints.

Example 62 Deletions due to infobox constraints. Consider the update u_1 inserting an alternative foaf:name value for an existing cleric (cf. the mapping in Ex. 61). The infobox key $\mathsf{IP} \to \mathsf{V}$ would deprecate this, since there is only one infobox property matching foaf:name. Therefore, all updates in $\{\![u_1^{\mathbf{W}}]\!\}$ will extend u_1 with the deletion of the old name.

Insertions and many-to-many property matches. Several Wiki properties are mapped to the same DBpedia property and the insertion cannot be uniquely resolved. E.g., infoboxes of football players in the English Wikipedia have the properties 'full name', 'name' and 'short name' all mapped to foaf:name.

Deletions with conditional mappings. Triples generated by a conditional mapping can be deleted either by removing the corresponding Wiki property or by modifying the Infobox property so that the condition is no longer satisfied. E.g., in Ex. 61, deleting the triple predecessor (Nicholas_II, Alexander_II) can be done either by unsetting the infobox property 'prédécesseur pape' or the property 'titre' used in the condition.

The above considerations suggest that despite the syntactic restrictions of the DBpedia mappings, the problem of update translation is hard in the worst case. Furthermore, numerous translations of an ABox update often exist (exponentially many in the size of the mapping: e.g., each n-to-m property match increases the total size of possible translations by the factor of mn). Due to the interplay between the mapping conditions

and TBox axioms a complete solution of the OBDM problem, presenting and explaining to the user *all possible ways* of accommodating an arbitrary update is not practical. Our pragmatic approach to the problem is described next.

6.3 Pragmatic DBpedia OBDM

Updates in the presence of constraints and mappings over a curated data source such as DBpedia are not likely to happen in a fully atomatic mode. Thus, rather than striving to define a set of formal principles to compare particular update implementations (akin, e.g. belief revision postulates) we focus on another aspect of update translation, especially important in collaborative and community-oriented settings, where adhering to standard practices and rules is crucial. Namely, we look for *most customary* ways to accommodate a change. For insertions, data evidence can be obtained from the actual data, whereas for deletions, additional logs are typically required. For all kinds of updates, we use a special kind of log, which we call *update resolution pattern*, recording the "shape" of each update command (e.g. *inserting a* birthPlace *DBpedia property of a* Pope *instance, where the Infobox property 'lieu de naissance' is already present. Delete the existing property and add the property 'lieu naissance' with the new value*).

To decide on the update pattern, when several alternatives are possible, we try to derive most customary ways of mapping objects of same classes from the existing data, rather than applying some principled belief revision semantics. E.g., when updating the birth place, we look at the usage statistics of the Infobox properties 'lieu naissance' and 'lieu de naissance' and choose the one used most often. If most infoboxes have both, we will not delete the already existing property but just add a second one. This way, we might resolve a DBpedia's foaf:name as two infobox properties (e.g., 'name' and 'full name') at once if most existing records of a given type follow this pattern, even if it would contradict the minimal change principle which typically governs belief revision.

A translation procedure we discuss next proceeds essentially on the best effort basis, exploring the most likely update accommodations and facilitating reuse of standard practices through update resolution patterns. It takes a SPARQL update and transforms it into a set of Infobox updates for the user to apply and save as an update resolution pattern. The source code of the system is openly available[4].

6.3.1 Update Translation Steps

From the very beginning, we turn our SPARQL update into a set of ground atoms, which are then grouped by subject (corresponding to the Wikipedia page). The idea of our update translation procedure is to create or to re-use existing update patterns for each grouped update extracted from the user input. A user update request related to a particular Wikipedia page (DBpedia entries grouped by a common subject) becomes a core pattern, which gives rise to a number of possible translations as a wiki insert.

[4]https://github.com/aahmeti/DBpedia-SUE, a screencast is available at https://goo.gl/BQhDYf

For each translation, the mapping and the TBox constraints are applied, in order to see which further atoms have to be added and if there are inconsistencies with the pre-existing facts. All such inconsistencies are removed, resulting in a further update, giving rise to an update resolution pattern nested within the root one, and the translation process proceeds recursively.

Pruning is essential in this process, since resolution patterns can sprout actively (e.g., some DBpedia properties are mapped to tens of Wikipedia ones). Potentially non-terminating, with the current DBpedia mappings inconsistencies can typically be resolved within the scope of one or two subjects (Wikipedia pages), and thus pattern trees resulting from this process are not deep. The reason is that functionality is currently only used for data properties, and only very few properties are declared disjoint.

The update translation algorithm, explained in more detail below, proceeds on a best effort basis, only following a single top ambiguity resolution option. If not terminated within some predefined number of steps, the translation aborts (not shown in the pseudocode).

The translation steps are outlined in Alg. 6.1. We keep a queue of pending and a list of already processed updates (line 1). First, we check if the update is inconsistent (line 2). In such case, we apply the *brave*[5] (cf. Chapter 5) semantics on the ABox side (line 3), which comes down to the principle "add all consequences and remove all causes", which may result in new triples to be added and deleted to resolve the inconsistency: these become pending updates (line 5), otherwise we initialize the list of pending elements with the initial update. For each triple in the list of such pending updates, we mark it as processed (line 8), get the Wiki page related to its subject (line 9) and its wikipedia templates (line 10). Next, we proceed to invert the wikipedia mappings (lines 11 and 12), as follows. The function *getCandidateWikiUpdate_def(triple,templates,e)* first applies the wikipedia updates e (produced in the previous iteration) to the wikipedia `templates`, in order to maintain a consistent view of the Wiki information. Then, we start with processing unconditional mappings and, for each one, checks if its deletion (resp. insertion) effectively deletes (resp. insert) the given `triple`. Note that, due to the mentioned n-to-1 correspondences, several alternatives could be present, hence the result `opt` may be a set of alternative candidates.

We repeat the procedure for the conditional mappings (line 12) noting that, in this case, the alternative are related to different branches whose conditions could be set/unset to satisfy the given `triple`. Then, the purpose of the operation is inspected. If the operation is delete (line 13), the n-to-1 alternatives can be disambiguated, given that we delete all of them whose resultant value matches the object of the `triple` to be deleted. As for the conditional branches, we define a function *chooseBranch* to decide which branch is set/unset. Given that the conditions highly depends on text constrains (e.g. they condition checks if one property contains certain text), we leave the decision to the user. In future work, we consider to apply additional heuristics to alleviate user decisions

[5] In a similar manner, we could also apply *cautious* or *fainthearted* semantics.

Algorithm 6.1: DBpedia update translation algorithm

Input: DBpedia update $u = (u^-, u^+)$
Output: Wiki update $e = (e^-, e^+)$

1 $Pending := \{\}$, $Processed := \{\}$
2 **if** $(\mathcal{M}(I) \setminus u^- \cup u^+) \models \bot$ **then**
3 | // use mapping-agnostic ABox conflict resolution semantics (Chapter 5) to $u := \mathbf{S}em_{brave}(\mathcal{M}(I) \cup u^+)$
4 **end**
5 $Pending.push(u^+, \text{“add”})$, $Pending.push(u^-, \text{“del”})$
6 **while** $(!Pending.isEmpty())$ **do**
7 | $(triple, operation) = Pending.pop()$
8 | $Processed.add(triple, operation)$
9 | $wiki = getWikiPage(triple.getSubject())$
10 | $templates[] = getTemplates(wiki)$
11 | $opt := getCandidateWikiUpdate\_def(triple, templates, e)$
12 | $cond := getCandidateWikiUpdate\_conditions(triple, templates, e)$
13 | **if** $(operation = \text{“del”})$ **then**
14 | | // choose between conditional options $(r^-, r^+) := opt \cup chooseBranch(cond)$
15 | **else**
16 | | // choose property mapping resp. condition branch $(r^-, r^+) := policyResolution(opt) \cup chooseBranch(cond)$
17 | **end**
18 | $(e^-, e^+) := (e^- \cup r^-, e^+ \cup r^+)$ // add to Wiki update results
19 | **if** $(\mathcal{M}(I \setminus e^- \cup e^+) \cup \mathcal{T}) \models \bot$ **then**
20 | | $(u_1^-, u_1^+) := \mathbf{S}em_{brave}(\mathcal{M}(I \cup r^+))$
21 | | $Pending.push(u_1^+, \text{“add”})$, $Pending.push(u_1^-, \text{“del”})$
22 | | $Pending.removeDuplicates()$
23 | | $Pending := Pending \setminus Procesed$ // disregard already processed updates
24 | **end**
25 **end**

on conditional branches. In turn, if the operation is insert (line 16), we maintain the same policy for the conditions, but the apply a policy resolution translation to try to disambiguate the n-to-1 alternatives to insert. This policy is further explained below, in Section 6.3.2. Finally, the Wiki updates are added to the previous considered updates (line 18) and we check if the resultant update up to now is inconsistent (line 19). In such case, we apply the `brave` semantics (cf. Chapter 5) (line 20), and add its recommended insertions and deletions to the pending query (line 21), removing potential duplicates (line 22) and disregarding already processed updates (line 23).

Finally, the resulting Wiki revisions are output to the user. Future work also considers to

Infobox	Subjects	Ambiguous n-1 mapping	
		Wikipedia prop.	Dbpedia prop.
Settlement	369,024	*area_total_km2* *area_total_sq_mi* *area_total* *TotalArea_sq_mi*	dbp:areaTotal
Taxobox	293,715	*species* *subspecies* *variety* *species_group* *species_subgroup* *species_complex*	dbp:species
Person	168,372	*website* *homepage*	foaf:homepage
Football biography	128,602	*name* *fullname* *playername*	foaf:name
Film	106,254	*screenplay* *writer*	dbp:writer

Table 62: Examples of n-to-1 alternatives in DBpedia (English) mappings.

group triples by subject. This allows the updates to be accommodated on page-by-page basis, and try to accommodate several properties at the same time.

6.3.2 Update Resolution Policies

Given the large number of possible translations of an update, potentially resulting in different clash patterns, an update can be translated in various ways, from which the user must select one. The crucial issue here is that the number of choices can be too large even for a very simple update, and that updates can cause side effects outlined in the previous section.

Here, we consider *update resolution policies* aimed at reducing the number of options for the user in the specific case of n-to-1 alternatives to insert. We currently consider two different alternatives in accordance with some concise principles, namely *infobox-frequency-first* and *similar-subject-first*.

We exemplify the application of such policies looking at the ambiguities in the top 10 most used Infoboxes[6]. In particular, we find and inspect the ambiguities in 'Settlement', 'Taxobox', 'Person', 'Football biography' and 'Film'. For the sake of clarity, we show a selection of the most representative ambiguities in Table 62, while other ambiguities in the infoboxes follow the same patterns. For instance, all 'name', 'fullname' and 'player name' in a 'football biography' infobox map to a foaf:name property. Table 62 also reports the number of subjects (i.e., wikipedia pages) of each infobox type, converted from the English Wikipedia.

Infobox-frequency-first. This policy considers that, for an insertion in a subject with an infobox W, resulting in a n-to-1 alternatives, we infer that the most likely accommodation would be the most frequent property in all the subjects with such infobox W,

[6]http://mappings.dbpedia.org/server/statistics/en/.

among all the alternatives not fulfilled in the subject we are currently updating. Statistics on frequent properties can be computed seamlessly, concurrently to the DBpedia conversion. Overall, this approximation could help users to inspect frequent properties for the update, so that rare or infrequent properties can be quickly discarded. In contrast, the approach may fail to guess the concrete purpose or real users, who may choose to accommodate different alternatives.

Similar-subject-first. The objective of this strategy is to refine the previous *Infobox-frequency-first* policy by delimiting a set of similar subjects for which the frequent properties are inspected. The reason of this strategy is that most of the properties in infoboxes are optional, so that different Wikipedia resources can, and often are, described with different levels of detail. Thus, finding "similar" subjects could effectively recommend more frequent patterns. For finding similar entities, we focus for the moment on a simple approach on sampling m subjects described with the same target infobox W and described with the same DBpedia property as the update u.

Recently Wikipedia partially shifted to another, structured datasource than infoboxes, namely, Wikidata. We note that the data model of Wikidata is different to DBpedia, nevertheless our approach that we will discuss in the following could potentially help in bridging between the two. In fact, we have developed a method to calculate similar entities in Wikidata, with the ultimate aim of providing a completeness measure for the Wikidata entities.

6.3.3 Assessing the Completeness of Wikidata Entities

Knowledge bases (KBs) such as Wikidata [VK14] or DBpedia [ABK+07] are becoming increasingly popular as structured sources of data, and are used in a variety of tasks such as structured search, question answering, or entity recognition, even though they are generally highly *incomplete* [RSN16]. In Wikidata, for instance, only 48% of politicians are member of a party, or only 0.02% of people do have a child. In particular, when incomplete KBs are combined with query languages that contain negation such as SPARQL, the result easily yields unsound answers [PFH06].

Understanding how complete KBs are on different aspects is important for KB curators so they know where to focus their efforts, and for consumers to know to which extent they can rely on a KB.

In this dissertation we propose to assess the relative completeness of entities in knowledge bases, based on comparing the extent of information with other similar entities. It is difficult to talk about the completeness of KBs because completeness can be investigated on various levels and with varying semantics. While it is relatively easy to understand when a knowledge base is complete for children of Obama (when Malia and Sasha are there), it is not clear what completeness of Obama himself, or of US politicians as a whole, could mean. Previous work on knowledge base completeness has focused on the lowest level, i.e., finding out when a subject is complete for a predicate (like Obama

for *child*) [GRAS17, MRN16, PDRN16], whereas more abstract levels have not been investigated so far.

In more detail, previous work on assessing KB completeness has focused on the level of subject-predicate pairs. [PDRN16] provides a plugin for Wikidata that allows to assert completeness for such pairs directly on the Wikidata website. [GRAS17] has used association rule mining for automatically determining complete pairs. [MRN16] used Wikipedia texts to mine the cardinalities of such pairs, using these cardinalities in turn to assess completeness. A recent survey paper, [Pau17], provides a comprehensive overview on the state-of-the-art KB refinement approaches aimed at improving the KB completeness. For more holistic descriptions of quality, Wikipedia has so-called status indicators (like "Featured article", "Good article"). For Wikidata, such indicators do not yet exist, but their introduction is planned.[7]

For basic granularities, such as children of Obama, as discussed in [GRAS17, MRN16, PDRN16], boolean completeness annotations generally suffice. In contrast, on the entity level, given that Wikidata contains over 2700 properties, of which 101 are used at least 1000 times for the class *human*, containing further ill-defined properties such as *medical condition*, *notable work* and *participant of*, it is clear that boolean statements such as "Data about Obama is complete", or "Data about Trump is incomplete", are not meaningful.

To allow statements for entities, we thus propose to define a relative completeness measure. More specifically, we propose to compare the extent of information about an entity with the extent of information that is available for other, similar entities. Elaborating it further, for a designated entity we check its coverage of frequent properties, computed among similar entities. For instance, in assessing the completeness of Obama, we would compare the information available about him with that available for other politicians, while when assessing the completeness of Austria, we would compare with other countries.

There are three crucial components to this approach, (i) the definition of similar entities, (ii) the way how the extent of information is compared among similar entities, and (iii) the way how explanations are provided.

(i) For similarity, classes are a natural baseline, and class-like properties such as *occupation* allow a further refinement. Semantic similarity measures [RE03] could provide even better way to find similar entities.

(ii) Baselines for comparison could be counts of facts or properties, while better results can be expected if the relevance or importance of properties and facts is taken into account [DA16].

[7] https://en.wikipedia.org/wiki/Wikipedia:Wikidata#Article_status_indicators

(iii) The way explanations are generated depends highly on the choices made for (i) and (ii), and will in turn impact usability for knowledge base authors and users. We may expect a tradeoff between accuracy and complexity, i.e., more complex choices may lead to more accurate assertions, which however are harder to explain, thus not necessarily increasing usability.

In the next chapter, we discuss the implementation and experiments for the update semantics discussed in Chapter 4, Chapter 5 and the present one (Chapter 6).

CHAPTER 7

Implementation and Experiments

In this chapter, following "from theory to practice" methodology (as discussed in Sec. 1.2), we show the feasibility of the ABox update semantics discussed in Chapters 4-6 by implementing them in practice, starting from the least expressive towards the most expressive fragment. For each one of them, we also provide experimental evaluations. Note that we do not discuss the implementation of TBox update semantics elaborated in Chapter 4.

7.1 Prototype and Experiments – *DL-Lite*$_{\text{rdfs}}$

We have implemented all of the *DL-Lite*$_{\text{RDFS}}$ update semantics discussed (cf. Chapter 4) except $\mathbf{Sem}_3^{mat}$, in Jena TDB[1] as a triple store that both implements the latest SPARQL 1.1 specification and supports rule based materialisation: our focus here was to use an existing store that allows us to implement the different semantics with its on-board features; that is, for computing $mat(G)$ we use the on-board, forward-chaining materialisation in Jena.[2] For each one of the semantics where fresh variables are introduced, we implemented them with two concrete variants of P_d^{fvars}. In the first variant, we replace $?v$ a rdfs:Resource by {{$?v$ $?v_p$ $?v_o$} UNION {$?v_s$ $?v$ $?v_o$} UNION {$?v_s$ $?v_p$ $?v$}}, to achieve a pattern that binds $?v$ to every possible term in G. This is not very efficient. In fact, note that P_d^{fvars} is needed just to bind fresh variables $?v$ (corresponding to '_' in Table 21) in patterns $P_{?v}$ of the form $P(x, ?v)$ or $P(?v, x)$ in the rewritten DELETE clause. Thus, we can equally use $P_d^{fvars'} = \{\texttt{OPTIONAL}\{\bigcup_{?v \in Var(P_d^{caus}) \setminus Var(P_d)}\} P_{?v}\}$. We denote implementations using the latter variant $\mathbf{Sem}_{2'}^{mat}$ and $\mathbf{Sem}_{1'}^{red}$, respectively.

As for reduced semantics, remarkably, for the restricted set of ABox rules in Fig. 23 and assuming an acyclic TBox, we can actually compute $red(G)$ also by "on-board" means of SPARQL 1.1 compliant triple stores, namely by using SPARQL 1.1 Update

[1] http://jena.apache.org/documentation/tdb/

[2] http://jena.apache.org/documentation/inference/

in combination with SPARQL 1.1 property paths [HS13, Section 9] with the following update:

```
DELETE  { ?S1 a ?D1. ?S2 a ?C2. ?S3 ?Q3 ?O3. ?O4 a ?C4. }
WHERE   {{ ?C1 sc+ ?D1. ?S1 a ?C1. }
   UNION { ?P2 dom/sc* ?C2. ?S2 ?P2 ?O2. }
   UNION { ?P3 sp+ ?Q3. ?S3 ?P3 ?O3. }
   UNION { ?P4 rng/sc* ?C4. ?S4 ?P4 ?O4. }}
```

We emphasise that performance results should be understood as providing a general indication of feasibility of implementing these semantics in existing stores rather than actual benchmarking: on the one hand, the different semantics are not comparable in terms of performance benchmarking, since they provide different results; on the other hand, for instance, we only use naive re-materialisation provided by the triple store in our prototype, instead of optimised versions of DRed, such as [UMJ+13].

For initial experiments we have used data generated by the LUBM [GPH05] generator for 5, 10 and 15 Universities, which correspond to different ABox sizes merged together with an RDFS version of the LUBM ontology as TBox; this version of LUBM has no complex restrictions on roles, no transitive roles, no inverse roles, and no equality axioms, and axioms of type $A \sqsubseteq B \sqcap C$ are split into two axioms $A \sqsubseteq B$ and $A \sqsubseteq C$. For instance, axioms $Chair \equiv Person \sqcap \exists headOf.Department$, $Chair \sqsubseteq Person$ and $Chair \equiv Person \sqcap \exists worksFor.Organization$ are stripped-down to the following RDFS axioms – as generic as possible and this is because properties are used with different ranges in restrictions (mandatory participation axioms):

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

ub:Chair a rdfs:Class;
rdfs:subClassOf ub:Person .

ub:headOf a rdf:Property;
rdfs:domain ub:Person;
rdfs:range ub:Organization;
rdfs:subPropertyOf ub:worksFor.

ub:worksFor a rdf:Property;
rdfs:domain ub:Person;
rdfs:range ub:Organization;
rdfs:subPropertyOf ub:memberOf.
```

Here it is an excerpt from the resulting ontology showing some of the main concepts:

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

```
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

ub:Employee a rdfs:Class;
rdfs:label "Employee";
rdfs:subClassOf ub:Person.

ub:Department a rdfs:Class;
rdfs:label "university department";
rdfs:subClassOf ub:Organization.

ub:Student a rdfs:Class;
rdfs:label "student";
rdfs:subClassOf ub:Person.

ub:UndergraduateStudent a rdfs:Class;
rdfs:label "undergraduate student";
rdfs:subClassOf ub:Student.

ub:GraduateCourse a rdfs:Class;
rdfs:label "Graduate Level Courses";
rdfs:subClassOf ub:Course.

ub:GraduateStudent a rdfs:Class;
rdfs:label "graduate student";
rdfs:subClassOf ub:Student.

ub:Book a rdfs:Class;
rdfs:label "book";
rdfs:subClassOf ub:Publication.

ub:Article a rdfs:Class;
rdfs:label "article";
rdfs:subClassOf ub:Publication.

ub:FullProfessor a rdfs:Class;
rdfs:label "full professor";
rdfs:subClassOf ub:Professor.

ub:undergraduateDegreeFrom a rdf:Property;
rdfs:domain ub:Person;
rdfs:label "has an undergraduate degree from";
rdfs:range ub:University;
rdfs:subPropertyOf ub:degreeFrom.

ub:worksFor a rdf:Property;
rdfs:domain ub:Person;
rdfs:label "Works For";
rdfs:range ub:Organization;
```

```
rdfs:subPropertyOf ub:memberOf.

ub:takesCourse a rdf:Property;
rdfs:domain ub:Student;
rdfs:label "is taking";
rdfs:range ub:Course.

ub:publicationAuthor a rdf:Property;
rdfs:domain ub:Publication;
rdfs:label "was written by";
rdfs:range ub:Person.

ub:headOf a rdf:Property;
rdfs:domain ub:Person;
rdfs:label "is the head of";
rdfs:range ub:Organization;
rdfs:subPropertyOf ub:worksFor.

ub:memberOf a rdf:Property;
rdfs:domain ub:Person;
rdfs:label "member of";
rdfs:range ub:Organization.

ub:member a rdf:Property;
rdfs:domain ub:Organization;
rdfs:label "has as a member";
rdfs:range ub:Person.

ub:mastersDegreeFrom a rdf:Property;
rdfs:domain ub:Person;
rdfs:label "has a masters degree from";
rdfs:range ub:University;
rdfs:subPropertyOf ub:degreeFrom.

ub:degreeFrom a rdf:Property;
rdfs:domain ub:Person;
rdfs:label "has a degree from";
rdfs:range ub:University.

ub:doctoralDegreeFrom a rdf:Property;
rdfs:domain ub:Person;
rdfs:label "has a doctoral degree from";
rdfs:range ub:University;
rdfs:subPropertyOf ub:degreeFrom.

ub:teacherOf a rdf:Property;
rdfs:domain ub:Faculty;
rdfs:label "teaches";
rdfs:range ub:Course.
```

```
ub:advisor a rdf:Property;
rdfs:domain ub:Person;
rdfs:label "is being advised by";
rdfs:range ub:Professor.

ub:hasAlumnus a rdf:Property;
rdfs:domain ub:University;
rdfs:label "has as an alumnus";
rdfs:range ub:Person.

...
```

Besides, we have designed a set of 7 different ABox updates in order to compare the proposed mat-preserving and red-preserving semantics.

Update #1. Transfer all students of course 0 from undergraduate students to graduate students.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

DELETE {?X rdf:type ub:UndergraduateStudent .}
INSERT {?X rdf:type ub:GraduateStudent . }
WHERE
{
    ?X ub:takesCourse <http://www.Department0.University0.edu/Course0> .
}
```

Update #2. Slight variation of #1: Transfer all students of course 0 to graduate students.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

DELETE {?X rdf:type ub:Student .}
INSERT {?X rdf:type ub:GraduateStudent . }
WHERE
{
    ?X ub:takesCourse <http://www.Department0.University0.edu/Course0> .
}
```

Update #3. Professor 11 moves to department 1 of university 1 and takes along all of her students. (Note that worksFor is a subproperty of memberOf, so also the memberOf relation for Professor 11 should be deleted for semantics where the effects are removed).

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
```

```
DELETE {<http://www.Department0.University0.edu/AssociateProfessor11>
                        ub:worksFor ?Y . ?Z ub:memberOf ?Y . }
INSERT {<http://www.Department1.University1.edu/AssociateProfessor11>
                        ub:headOf <http://www.Department1.University1.edu> .
                ?Z ub:memberOf <http://www.Department1.University1.edu>  . }
WHERE
{
    <http://www.Department0.University0.edu/AssociateProfessor11>
                ub:worksFor ?Y .
    ?Y rdf:type ub:Department .
    ?Z ub:advisor <http://www.Department0.University0.edu/AssociateProfessor11>
.
    ?Z rdf:type ub:GraduateStudent .
    ?Z ub:memberOf ?Y .
}
```

Update #4. Auto-register all graduate students to all of the courses taught by their advisor.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

INSERT {?X ub:takesCourse ?W .}
WHERE
{
    ?X rdf:type ub:GraduateStudent .
    ?Z ub:teacherOf ?W .
    ?X ub:advisor ?Z .
}
```

Update #5. Enforce the policy that a student can only have one undergraduate degree per university and may not re-register as an undergrad student if she already has any degree from that university. Additionally, mark all the students with a degree as alumni.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

DELETE {?X rdf:type ub:UndergraduateStudent .}
INSERT {?Y ub:hasAlumnus ?X.}
WHERE
{
    OPTIONAL {?X ub:undergraduateDegreeFrom ?Y . }
    OPTIONAL {?X ub:mastersDegreeFrom ?Y . }
    OPTIONAL {?X ub:doctoralDegreeFrom ?Y . }
}
```

Update #6. Variant of #5 which uses UNION instead of OPTIONAL. Enforce the policy that a student can only have one undergraduate degree per University and may not re-register as an undergrad student if she already has any degree from that university. Additionally, mark all the students with a degree as alumni.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

DELETE {?X rdf:type ub:UndergraduateStudent .}
INSERT {?Y ub:hasAlumnus ?X.}
WHERE
{
    {?X ub:undergraduateDegreeFrom ?Y . }
    UNION
    {?X ub:mastersDegreeFrom ?Y . }
    UNION
    {?X ub:doctoralDegreeFrom ?Y . }
}
```

Update #7. Assistant Professor 0 got some funding and wants to employ all her not-already employed undergraduate co-authors as graduate students.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

DELETE {?Y rdf:type ub:UndergraduateStudent . }
INSERT {?Y rdf:type ub:GraduateStudent . ?Y ub:advisor
                <http://www.Department0.University0.edu/AssistantProfessor0> .
        ?Y ub:worksFor ?Z .}
WHERE
{
?X ub:publicationAuthor
                <http://www.Department0.University0.edu/AssistantProfessor0> .
?X ub:publicationAuthor ?Y .
FILTER NOT EXISTS {?Y rdf:type ub:Employee . }
<http://www.Department0.University0.edu/AssistantProfessor0> ub:worksFor ?Z .
}
```

Both the prototype, as well as files containing the data, ontology, and the updates used for experiments are available on a dedicated Web page.[3]

We first compared, for each update semantics, the time elapsed for rewriting and executing the update. Secondly, in order to compare mat-preserving and red-preserving semantics, we also need to take into account that red-preserving semantics imply additional effort on subsequent querying, since rewriting is required (cf. Prop. 4). In order to reflect this, we also measured the aggregated times for executing an update and subsequently processing the standard 14 LUBM benchmark queries in sequence. The evaluation of the respective update plus 14 queries merged together for different dataset sizes are reported in Table 71, Table 72 and Table 73 respectively.

In general, among the mat-preserving semantics, the semantics implementable in terms of rewriting ($\mathbf{Sem}_2^{mat}$) perform better than those that need rematerialisation ($\mathbf{Sem}_{1a,b}^{mat}$), as could be expected. There might be potential for improvement here on the latter, when using tailored implementations of DRed. Also, for both mat-preserving ($\mathbf{Sem}_{2'}^{mat}$) and red-preserving ($\mathbf{Sem}_{1'}^{red}$) semantics that rely on rewritings for deleting causes, the optimi-

[3] http://dbai.tuwien.ac.at/user/ahmeti/sparqlupdate-rewriter/

	Update #1	Update #2	Update #3	Update #4	Update #5	Update #6	Update #7
$\mathbf{Sem}_0^{mat}$	26,7	23,2	23,1	223,7	29,0	24,9	24,0
$\mathbf{Sem}_{1a}^{mat}$	30,2	29,9	30,6	56,7	39,8	41,3	30,9
$\mathbf{Sem}_{1b}^{mat}$	22,4	23,8	28,7	35,5	42,0	35,4	33,2
$\mathbf{Sem}_2^{mat}$	0,1	1454,1	0,1	271,2	1,1	0,9	0,1
$\mathbf{Sem}_{2'}^{mat}$	0,1	0,1	0,1	278,5	1,3	0,9	0,1
$\mathbf{Sem}_0^{red}$	17,8	9,8	10,0	179,2	12,1	13,6	11,0
$\mathbf{Sem}_1^{red}$	17,8		10,0	179,2	12,1	13,6	11,0
$\mathbf{Sem}_{1'}^{red}$	17,8	16,3	10,0	179,2	12,1	13,6	11,0

Table 71: Evaluation results in seconds (s) for LUBM 5 in the context of $\mathit{DL\text{-}Lite}_{\text{RDFS}}$ (an empty cell represents a run-time exception).

sation of using variant $P_d^{fvars'}$ instead of P_d^{fvars} paid off for our queries. This can be seen especially in Update #2 where the performance gain is significant for both $\mathbf{Sem}_{2'}^{mat}$ and $\mathbf{Sem}_{1'}^{red}$, especially for bigger LUBM datasets where results without optimisation ended with run-time exceptions. The same observation, albeit to a lesser degree, can be also drawn for Update #4. For the other updates, the improvement is not significant due to the nature of the update, i.e., these updates do not have causes to be computed. As for a comparison between mat-preserving vs. red-preserving, in our experiments re-reduction upon updates seems quite affordable, whereas the additionally needed query rewriting for subsequent query answering does not add dramatic costs. Thus, we believe that, depending on the use case, keeping reduced stores upon updates is a feasible and potentially useful strategy, particularly since—as shown above—$\mathit{red}(G)$ can be implemented with off-the-shelf features of SPARQL 1.1.

While further optimisations, and implementations in different triple stores are beyond the scope of this dissertation, the experiments confirm our expectations so far.

7.2 Prototype and Experiments – $\mathit{DL\text{-}Lite}_{\mathbf{rdfs}_\neg}$

For each of the three semantics discussed in Chapter 5, we provided a preliminary implementation using the Jena API (`http://jena.apache.org`) and evaluated them

	Update #1	Update #2	Update #3	Update #4	Update #5	Update #6	Update #7
$\mathbf{Sem}_0^{mat}$	51,6	56,4	46,4	844,7	49	49,8	48,5
$\mathbf{Sem}_{1a}^{mat}$	67,4	68,7	66	125,2	92,7	104,4	74,5
$\mathbf{Sem}_{1b}^{mat}$	60,1	56,2	65,1	63,5	71	57,7	57,8
$\mathbf{Sem}_2^{mat}$	0,1		0,2	753,7	1,7	1,5	0,2
$\mathbf{Sem}_{2'}^{mat}$	0,1	0,2	0,2	725,1	1,7	1,5	0,1
$\mathbf{Sem}_0^{red}$	27,6	29	29,3		30,7	30	30,9
$\mathbf{Sem}_1^{red}$	27,6		29,3		30,7	30	30,9
$\mathbf{Sem}_{1'}^{red}$	27,6	24,1	29,3		30,7	30	30,9

Table 72: Evaluation results in seconds (s) for LUBM 10 in the context of $\mathit{DL\text{-}Lite}_{\text{RDFS}}$ (an empty cell represents a run-time exception).

	Update #1	Update #2	Update #3	Update #4	Update #5	Update #6	Update #7
$\mathbf{Sem}_0^{mat}$	83,7	86,1	89	2045,6	88,4	89,6	86,7
$\mathbf{Sem}_{1a}^{mat}$	110,9	116,4	108,6	284,9	187,1	197,9	122,1
$\mathbf{Sem}_{1b}^{mat}$	95,2	105,7	89	108,4	104,6	105,9	86,1
$\mathbf{Sem}_2^{mat}$	0,1		0,1	2231,3	2,5	2,3	0,2
$\mathbf{Sem}_{2'}^{mat}$	0,1	0,2	0,2	1669	2,3	2,3	0,2
$\mathbf{Sem}_0^{red}$	49,5	48,2	46,9		47,6	48,6	48,6
$\mathbf{Sem}_1^{red}$	49,5		46,9		47,6	48,6	48,6
$\mathbf{Sem}_{1'}^{red}$	49,5	53,1	46,9		47,6	48,6	48,6

Table 73: Evaluation results in seconds (s) for LUBM 15 in the context of *DL-Lite*$_{\text{RDFS}}$ (an empty cell represents a run-time exception).

against Jena TDB triple store which implements the latest SPARQL 1.1 specification. As before, for computing the initial materialisation of a triple store $mat(G)$ we rely on on-board, forward-chaining materialisation in Jena TDB using the minimal RDFS rules as in Fig. 23.

For our experiments, we used the data generated by the EUGen generator [LSTW13] of the size range of 5 to 50 Universities. We opted for using this generator as it extends the LUBM ontology [GPH05] with chains of subclasses, making the rewritings more challenging. In our case we have used the default of $i = 20$ subclasses for each LUBM concept (e.g., `Subj`i`Students`) and made such subclasses pairwise disjoint. Moreover, we have added more disjointness axioms where appropriate, e.g., `:AssociateProfessor` dw `:FullProfessor`. All these TBox axioms are merged with the reduced RDFS version of LUBM (cf. Sec. 7.1). Here it is an excerpt from the ontology that emphasises the new concepts:

```
prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix owl:   <http://www.w3.org/2002/07/owl#>
prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
prefix ub:    <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

ub:Subj1Course a rdfs:Class ;
rdfs:subClassOf ub:Course ;
owl:disjointWith ub:Subj2Course .

ub:Subj5Department a rdfs:Class ;
rdfs:subClassOf ub:Department ;
owl:disjointWith ub:Subj2Department .

ub:Subj3Professor a rdfs:Class ;
rdfs:subClassOf ub:Professor ;
owl:disjointWith ub:Subj4Professor .

ub:Subj10Student a rdfs:Class ;
rdfs:subClassOf ub:Student ;
owl:disjointWith ub:Subj2Student .
```

```
ub:Alumnus owl:disjointWith ub:UndergraduateStudent .
ub:Employee owl:disjointWith ub:Student .
ub:UndergraduateStudent owl:disjointWith ub:GraduateStudent .
ub:AssistantProfessor owl:disjointWith ub:AssociateProfessor .
ub:AssistantProfessor owl:disjointWith ub:FullProfessor .
ub:AssociateProfessor owl:disjointWith ub:FullProfessor .
ub:PostDoc owl:disjointWith ub:Professor .
ub:Lecturer owl:disjointWith ub:Professor .

ub:ConferencePaper owl:disjointWith ub:Book .
ub:ConferencePaper owl:disjointWith ub:Article .
ub:ConferencePaper owl:disjointWith ub:Manual .
ub:ConferencePaper owl:disjointWith ub:JournalArticle .
ub:ConferencePaper owl:disjointWith ub:Software .
ub:ConferencePaper owl:disjointWith ub:UnofficialPublication .
ub:ConferencePaper owl:disjointWith ub:TechnicalReport .

ub:Article owl:disjointWith ub:Book .
ub:Article owl:disjointWith ub:Manual .
...
```

To compare the experimental results with Sec. 7.1, for our experiments we adapted the seven updates from Sec. 7.1.

Update #1. Transfer all students of course 0 to graduate students.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

INSERT {?X rdf:type ub:GraduateStudent . }
WHERE
{
        ?X ub:takesCourse <http://www.Department1.University0.edu/Course0> .
}
```

Update #2. Variant of #1, transfer all students of course 43 to subject18 students.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

INSERT {?X rdf:type ub:Subj18Student .}
WHERE
{
        ?X ub:takesCourse <http://www.Department1.University0.edu/Course43> .
}
```

Update #3. Transfer all publications of Associate Professor 1 from books to articles.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

DELETE {?X rdf:type ub:Book . }
INSERT {?X rdf:type ub:Article .}
WHERE
{
        ?X ub:publicationAuthor
            <http://www.Department0.University0.edu/AssociateProfessor1> .
}
```

Update #4. Variant of #3, transfer all publications of graduate students from books to articles.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

DELETE {?X rdf:type ub:Book . }
INSERT {?X rdf:type ub:Article .}
WHERE
{
        ?X ub:publicationAuthor ?Y .
        ?Y a ub:GraduateStudent .
}
```

Update #5. Insert all personnel working for university 0 as full professors.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

INSERT {?X rdf:type ub:FullProfessor .}
WHERE
{
        ?X ub:worksFor <http://www.Department0.University0.edu> .
}
```

Update #6. Insert all students who have finished a degree as alumni.

```
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

INSERT {?Y ub:hasAlumnus ?X.}
WHERE
{
    OPTIONAL {?X ub:undergraduateDegreeFrom ?Y . }
    OPTIONAL {?X ub:mastersDegreeFrom ?Y . }
    OPTIONAL {?X ub:doctoralDegreeFrom ?Y . }
}
```

Update #7. Variant of #6 with UNIONs instead of OPTIONALs, insert all students who have finished a degree as alumni.

```
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

INSERT {?Y ub:hasAlumnus ?X.}
```

```
WHERE
{
    {?X ub:undergraduateDegreeFrom ?Y . }
    UNION
    {?X ub:mastersDegreeFrom ?Y . }
    UNION
    {?X ub:doctoralDegreeFrom ?Y . }
}
```

The prototype, as well as files containing the data, ontology, and the updates used for experiments, are made available on a dedicated Web page[4].

The results summarized in Table 74 (cf. on smaller datasets Table 75 and Table 76) show that the LUBM 50 dataset (507MB uncompressed, 8.7M triples after materialisation) can be handled in seconds on a quad-core Intel i7 3.20 GHz machine with 16 GB RAM. For each of the three semantics, we have compared the time elapsed for rewriting and for the evaluation of the resulting update. The last line in Table 74 is the evaluation time for the original, non-rewritten update. One can notice that brave semantics $\mathbf{Sem}^{mat}_{brave}$ is often the most expensive one, since it performs most modifications. When the number of inconsistent inserts is low though, the situation is different, and the brave semantics slightly outperforms the fainthearted semantics $\mathbf{Sem}^{mat}_{faint}$ (Update #6 and #7), due to the more complex checks in the WHERE clause produced by Alg. 5.5. For the cautious semantics $\mathbf{Sem}^{mat}_{caut}$, the numbers in the table are construction and evaluation time of the ASK query checking for the feasibility of update (cf. Alg. 5.4). In case this ASK query returns $False$, the runtime of brave semantics should be added in order to obtain the total runtime of the update. Update #4 demonstrates that $\mathbf{Sem}^{mat}_{caut}$ can perform significantly worse than $\mathbf{Sem}^{mat}_{faint}$ when the number of instantiations in the original WHERE clause is high. This is because the ASK query in $\mathbf{Sem}^{mat}_{caut}$ looks for instantiations of the WHERE clause which can lead to clashes with the existing tuples (using a conjunctive condition), whereas $\mathbf{Sem}^{mat}_{faint}$ reduces the set of solutions of the original WHERE clause using MINUS, which is apparently more efficient in the Apache TDB.

	Update #1	Update #2	Update #3	Update #4	Update #5	Update #6	Update #7
$\mathbf{Sem}^{mat}_{brave}$	12,4	14,8	0,1	22,1	46,0	15,3	13,6
$\mathbf{Sem}^{mat}_{caut}$	0,3	0,2	0,2	44,0	0,2	3,9	2,3
$\mathbf{Sem}^{mat}_{faint}$	2,2	2,8	0,01	17,4	3,3	16,7	15,3
Original	0,2	0,2	0,2	10,2	0,2	6,6	5,4

Table 74: Evaluation results in seconds for LUBM 50 in the context of $DL\text{-}Lite_{\text{RDFS}\neg}$

[4] http://dbai.tuwien.ac.at/user/ahmeti/sparqlupdate-inconsistency-resolver/

	Update #1	Update #2	Update #3	Update #4	Update #5	Update #6	Update #7
$\mathbf{Sem}^{mat}_{brave}$	2,6	1,9	0,1	2,2	4,9	1,8	1,4
$\mathbf{Sem}^{mat}_{caut}$	0,2	0,2	0,3	5,1	0,2	1,0	0,4
$\mathbf{Sem}^{mat}_{faint}$	0,3	0,3	0,1	1,6	0,3	1,8	1,6
Original	0,2	0,2	0,2	1,2	0,2	1,5	0,8

Table 75: Evaluation results in seconds for LUBM 5 in the context of $DL\text{-}Lite_{\text{RDFS}\neg}$

7.3 Prototype and Experiments – (OWL) DBpedia

We have developed a proof of concept implementation that was designed to address the challenge of updating Wikipedia infoboxes via SPARQL 1.1 updates on DBpedia as discussed in Chapter 6. The current prototype, referred to as the DBpedia SPARQL Update Endpoint (DBpedia-SUE), provides a high-level façade to resolve the SPARQL updates and accommodate them in Wikipedia. The framework is developed in Scala[5] and makes use of the general components of the DBpedia Extraction Framework.

The architecture of our system is depicted in Fig. 71 and consists of several modules and services, explained in the following.

Restful API: This component is responsible for the interaction with the user. DBpedia-SUE can be accessed by the Web User Interface (UI) or querying via RESTful operations so that third-party applications can be built on top of our proposal, translating SPARQL Updates on DBpedia to the underlying Wiki content.

Fig. 72 shows an example on the Web UI of DBpedia-SUE. First, the user poses a SPARQL update query on DBpedia and selects the update semantics to be applied in case of inconsistencies (brave, cautious or fainthearted). Then, the user runs the query to get the potential Wikipedia results, i.e., the wikipedia properties to be added and deleted, and the resultant infobox text to be directly used within Wikipedia. When several alternatives are available to accommodate the update in wikipedia, these are presented to the user (*OPT* tabs in Fig. 72). The UI also provides the concrete DBpedia triples (added and inserted) after applying the user query and resolving the potential inconsistencies.

Consistency Checker Once the user runs the SPARQL update query, this component is aimed at checking its consistency. As a first step, if variables are present in the user

	Update #1	Update #2	Update #3	Update #4	Update #5	Update #6	Update #7
$\mathbf{Sem}^{mat}_{brave}$	3,3	2,9	0,1	4,2	9,5	2,6	2,8
$\mathbf{Sem}^{mat}_{caut}$	0,2	0,3	0,2	8,4	0,2	1,2	0,5
$\mathbf{Sem}^{mat}_{faint}$	0,4	0,4	0,1	3,1	0,3	3,4	2,4
Original	0,2	0,2	0,2	2,5	0,2	1,6	1,4

Table 76: Evaluation results in seconds for LUBM 10 in the context of $DL\text{-}Lite_{\text{RDFS}\neg}$

[5] http://www.scala-lang.org/

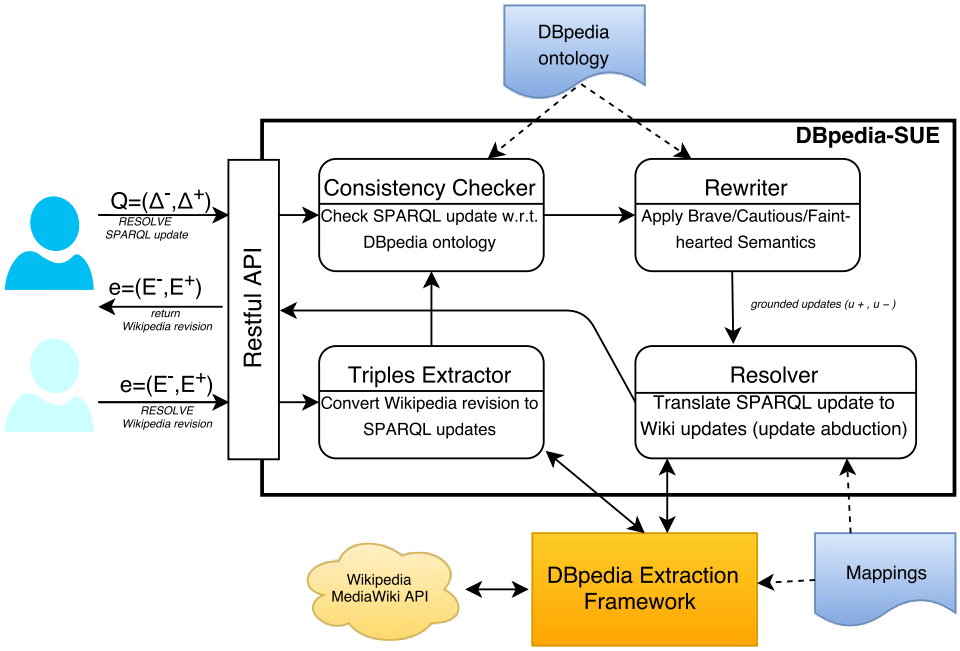


Figure 71: DBpedia-SUE architecture.

query, this component evaluates it against the DBpedia Live instance in order to produce a set of ground updates (triples with no variables). Then, it makes use of the most recent DBpedia ontology[6], whose formalization is provided in Sec. 6.1.

Rewriter. If the previous consistency checker reveals potential inconsistencies in the user query, then this component applies the SPARQL update `brave` semantics—cf. Chapter 5 for details—to solve potential clashes w.r.t. the DBpedia data.

Resolver. This component takes as input a set of grounded updates from the previous steps and translates them to Wiki updates. The details of update translation have been discussed in Sec. 6.3.1.

Triples Extractor. A user can validate Wikipedia revisions to check and resolve potential inconsistencies arising after the application of the revisions[7]. In this case, our framework—via the DBpedia Extraction Framework—first extracts the DBpedia triples emerging from the revisions. Then, these triples are treated as a user update query, following the aforementioned workflow.

[6] http://wiki.dbpedia.org/Downloads

[7] For instance, the application of a Wikipedia revision can activate/deactivate conditional mappings, whose extraction (i.e. the translation to DBpedia) may introduce novel inconsistencies.

Figure 72: DBpedia-SUE User Interface.

For experimental evaluations we have used the two update resolution policies *Infobox-frequency-first* and *Similar-subject-first*, which are used to reduce the number of options in case of INSERT alternatives.

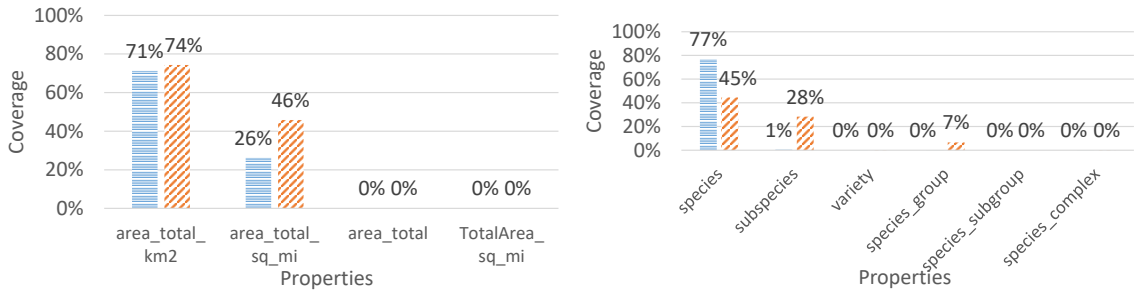
Infobox-frequency-first. Fig. 73 evaluates the distribution of frequencies of the Wikipedia properties involved in n-1 mappings from Table 62, considering all the subjects in the infobox (series *Infobox-frequency-first*). Results show that the application of this policy can certainly filter out infrequent property candidates, but it may require further elaboration for a more informed recommendation, specially in those cases in which the property is not extensively used in the infoboxes. For instance, all properties with no or marginal presence can be discarded, such as 'area_total' and 'TotalArea_sq_mi' in 'Settlement' (Fig. 73 (a)), 'variety', 'species_group', 'species_subgroup' and 'species_complex' in 'Taxobox' (Fig. 73 (b)), 'homepage' in 'Person' and 'playername' in 'Football biography' (Fig. 73 (d)). In turn, some properties are much more represented than others, and shall be the first ranked suggestion when inserting an ambiguous mapping. This is the case of most of the infoboxes, such as the frequent 'area_total_km2' property in 'Settlement', 'species' in 'Taxobox', 'website' in 'Person', and 'writer' in 'Film'. In contrast, only one case, 'Football biography', showed two properties that are almost equally distributed, with 'name' slightly more used than 'fullname'.

Similar-subject-first. Fig. 73 evaluates the distribution of property frequencies in such scenario (series *Similar-subject-first*), sampling $m = 1,000$ subjects of each infobox described the DBpedia property to be inserted (dbp:areaTotal, dbp:species, foaf:homepage, foaf:name or dbp:writer respectively). Results show that this policy allows the system to perform more informed decisions. For instance, in the 'Person' use case (Fig. 73 (d)), the 'homepage' property cannot be discarded (as suggested by the *Infobox-frequency-first* approach), given that a particular type of persons are more frequently associated with homepage instead of websites (e.g., those who are not related to a company). Similarly, in 'Taxobox' (Fig. 73 (b)), some particular species also include 'subspecies' and 'species_group', hence they should be included and ranked as potential accommodations for the user query.

7.3.1 Recoin (Relative Completeness Indicator)

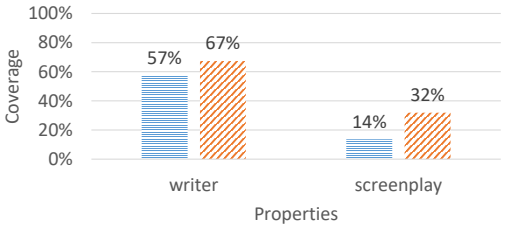
As discussed in Sec. 6.3.3, in the same spirit as 'similar-subject-first' policy elaborated in the previous section, a similar measure has been implemented for Wikidata. We have implemented a relative completeness indicator called *Recoin* in Wikidata.[8] It is provided as user script, i.e., logged in Wikimedia users can enable it in a user configuration file. It consists of two components. The core component, which adds a relative completeness indicator to the status indicator section of Wikidata articles, is shown in Fig. 74. The indicator is a color-coded progress bar, which can show 5 levels of completeness, ranging from "very detailed" to "very basic". An explanation module adds information about the relevant missing properties, based on which the completeness level is calculated. Further details about the architecture are on the tools website. It is currently available on the

[8] https://www.wikidata.org/wiki/User:Ls1g/Recoin

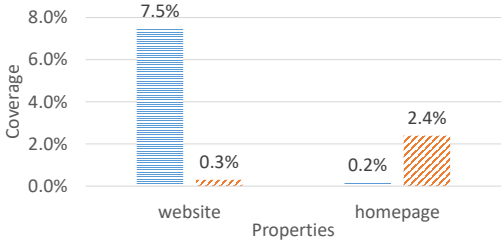


(a) Infobox Settlement

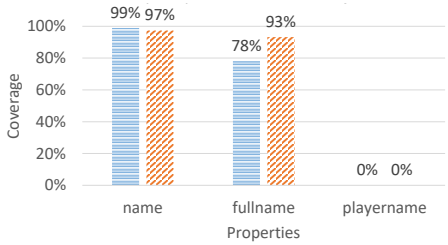
(b) Infobox Taxobox



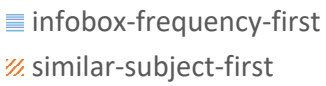
(c) Infobox Film



(d) Infobox Person



(e) Infobox Football bio.



(f) Legend for the charts (a)–(e).

Figure 73: Statistics obtained by *infobox-frequency-first* and *similar-subject-first* policies on four different infoboxes.

Wikidata pages of all *humans* that have a profession. Internally, the completeness level is computed as follows:

1. Each entity is compared with the set of all entities that have at least one profession in common.

2. For that set, the 50 most frequent properties are computed. The completeness level is then computed using fixed thresholds, i.e., if the entity has more than 40 out of these 50 properties, completeness is on the highest level, if it has between 30 and 40 of these properties, second highest level, and so on.

3. As explanation, the properties absent w.r.t. the comparison set are shown along with their frequency in the comparison set.

Figure 74: Recoin core module on the Wikidata page of Jimmy Wales.

The tool was made available to the Wikidata community on 15th of November, 2016. An expansion to all *humans* and other classes of entities are planned[9].

Some completeness levels computed by Recoin are for Obama 4 (detailed), for Trump 3 (fair), for Jimmy Wales 3 (fair), or for Dijkstra 2 (basic). While many levels appear reasonable (more popular entities are more complete, less popular ones less), others can only be understood using the explanations. The comparably low level for Jimmy Wales, for instance, is based on the fact that he misses properties such as *member of political party*, *position held* and *father*, which in the comparison set, exist for 10%, 8% and 6% of entities.

To further evaluate the levels computed by Recoin, in a crowdsourcing experiment, we compared a three-level scheme with levels that human annotators would give. Using 20 entities and 7 opinions per entity, we found that Recoin agreed in 60% of cases with the majority opinion, while in 25% it was off by one level, and in 15% off by two levels.

In the next chapter, we conclude the dissertation by discussing state-of-the-art approaches in detail and, finally we reflect on the achieved results and the issues that are left beyond the dissertation.

[9] As of 6th of December, 2017, Recoin has been expanded for all entities, for further details refer to [BRN18].

CHAPTER 8

Discussion and Outlook

In this chapter, we provide a discussion of related approaches, followed by the conclusions and future work. In the latter section, we conclude starting with reviewing each research question posed in Chapter 1, summarising the results, as well as discussing further related work. In the end, we discuss the issues that are left open, beyond this dissertation for future work.

8.1 Discussion of Other Related Approaches

In the following, we elaborate on the related studies on updates in OBDM, divided into ABox (cf. Sec. 8.1.1) and TBox (cf. Sec. 8.1.2) updates. We focus on the most prominent approaches, while other related works are briefly mentioned in the next section (cf. Sec. 8.2). Some of the approaches discussed next are theoretical, while others have a concrete implementation. Each individual approach is explained in detail as well as illustrated using an example. We also elaborate how each approach is different w.r.t. this dissertation, and whether we can use it for future work. In the end of each approach, we summarise based on OBDM components: integrity constraints, mappings language, ontology language and update language. To conclude, we provide a subsection (cf. Sec. 8.1.3) where we summarise the approaches we discussed versus this dissertation.

8.1.1 ABox Updates in Ontology-Based Data Management

Updating RDFS: from theory to practice [GHV11]

This approach proposes both theoretical and practical means of dealing with ABox updates in RDFS graphs. ABox and TBox are treated separately as in Description Logics, contrary to the conventional RDFS setting, where they are not distinguished. Updates comprise of only atomic deletes, as inserts do not pose any challenges in RDFS, i.e., merely boils down to the merge of the respective two graphs. In practice, deleting

ABox assertions is shown to be deterministic and finite, due to the following three reasons: (i) instances are treated separate from the schema; (ii) the nature of the rules used are the "minimal" RDFS (Fig. 23), i.e., "causes" can be found and traced back unambiguously; (iii) blank nodes are treated as constants.

In order to theoretically approximate the solutions obtained by the state after an update $\mathcal{U}$ over a graph $\mathcal{O}$, authors introduce the so-called *erase candidates*—denoted *ecand($\mathcal{O}$, $\mathcal{U}$)*—which contain the maximal closure of the graph $\mathcal{O}$ that does not entail $\mathcal{U}$. An erase candidate is representable as an RDFS graph and a disjunction (union) of them yields the complete solution, i.e., the answer of an update. For convenience reasons, the approach is extensively based upon the complement of erase candidates, so-called *delta candidates* – denoted as $dcand(\mathcal{O},\mathcal{U}) = (mat(\mathcal{O}) \setminus E) \colon E \in ecand(\mathcal{O},\mathcal{U})$. In this case, closure of the graph is not used for maintaining the graph itself, but rather it is used only for computing delta candidates.

Delta candidates contain the triples, deletion of which yields a graph, but does not entail the triple designated to be deleted. For computing *dcand*, practical algorithms for instances (and also schemas, cf. Sec. 8.1.2) are given. The algorithm for deleting instances is deterministic, and for each triple to be deleted computes all the "causes" that entail the respective triple. This operation can be reduced to computing reachability in a graph, thus can be computed in polynomial time:

- For triples of type `:jack a :Person .` the algorithm computes *dcand* – all the reachable triples, which entail the respective triple via

  "type" and "subclass" axioms (see Fig. 23) e.g., `:jack a :Employee .`

  "domain" and "range" axioms (see Fig. 23) e.g., `:jack :worksFor :marketing .`

- For triples of type `:jack :belongsTo :marketing .` the algorithm computes *dcand* – all the existent triples which entail via "subproperties" (see Fig. 23) e.g., `:jack :worksFor :marketing .`

Example 63 (Deleting RDFS instances in practice) Given an OBDM system $\langle \mathcal{O}_m, \mathcal{D} \rangle$, $\mathcal{O}_m = \langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$, such that $\mathcal{M}(\mathcal{D}) = \mathcal{A} = \{$`:jack a :Employee .` `:jack :worksFor :marketing`$\}$, merged together with the TBox $\mathcal{T}$ as defined in Ex. 3. Consider the update u:

```
DELETE { :jack a :Person }
```

then the algorithm computes the reachability for assertion `:jack a :Person`, computing u':

```
DELETE { :jack a :Person . :jack a :Employee . :jack :worksFor :marketing . }
```

■

Relation to our approach: The setting in this approach is similar to *DL-Lite*$_{\text{RDFS}}$ discussed in this dissertation; this can be observed by the three reasons mentioned above. For computing "causes", we compute the "reachability in the graph" as discussed here by leveraging *PerfectRef* algorithm, which in addition also takes into account DELETE/INSERT paired with a non-instantiated WHERE clause.

Integrity constraints: No integrity constraints at the sources are supported.
Mappings language: No mappings are supported.
Ontology language: Minimal RDFS - sc, sp, dom, rng (see Fig. 23).
Update language: DEL of atomic triples in a graph (INS can be implicitly supported).

ABox Evolution in DL-Lite [CKNZ10]

The approach deals with both model-based and formula-based evolution over the closed *DL-Lite*$_{\mathcal{FR}}$ ontologies[1]. The model-based semantics are shown to not be useful in practice and therein are not discussed, and this is primarily because the result of an update can not always be captured in DL-Lite. Since ABox and TBox are treated separately, in the case of ABox evolution, the TBox is considered protected and vice-versa. The approach proposes a new formula-based semantics called *bold semantics* (see Sec. 2.3 for disadvantages of Cross-Product and WIDTIO) that computes an unique and deterministic result. The *FastEvol* algorithm (Fig. 3, [CKNZ10]) that implements the bold semantics, for each fact to be inserted, checks whether there exists a set of facts in the ABox closure that might be inconsistent. The inconsistency in this logic can occur either from single ABox assertion, which is a member of unsatisfiable concept or role, or pair of ABox assertions contradicting either disjointness or functionality assertion of the TBox, directly following from Lemma 12 in [CKNZ10]. This lemma ensures that one can always resolve inconsistencies if one inserts ABox assertions, which are inconsistent w.r.t. TBox, consequently by deleting the old assertions in the knowledge base that contribute to the inconsistency. The algorithm processes each fact by always inserting the new knowledge "on hand" and its chase w.r.t. the TBox, and if there are inconsistencies after the update, the facts in the knowledge base are added to the set of facts to be deleted. The facts to be deleted, exploiting so-called *Weeding* algorithm (Fig. 2, [CKNZ10]), delete also all the "causes". The result of the algorithm is an unique and deterministic subset of the materialised/closed ABox.

Example 64 (Updating DL-Lite ABox axioms) Given the OBDM system $\langle \mathcal{O}_m, \mathcal{D} \rangle$, $\mathcal{O}_m = \langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$ where TBox $\mathcal{T} = \{Employee \sqsubseteq \exists worksFor, \exists worksFor \sqsubseteq Employee, Employee \sqsubseteq \neg Manager\}$, ABox $\mathcal{M}(\mathcal{D}) = \mathcal{A} = \{worksFor(jane, finance), Employee(jane)\}$ and the following update:

$u =$ **INSERT** $\{Manager(jane)\}$.

[1] A more recent version of the paper [ZKNC19], elaborates further on contraction and postulates.

According to *FastEvol*, first we incorporate the update u and its chase ("effects"). Then, we remove the inconsistencies and their causes (not including the mandatory participations of type $A \sqsubseteq \exists P$):

DELETE $\{Employee(jane)\}$.

After the update, a new axiom is derived w.r.t. TBox, so-called *role-constraining* formulas: $worksFor(x, finance), x \neq jane$, which is entailed neither from the old $\mathcal{O}_m$, nor from u.

This issue is resolved by the *CarefulEvol* algorithm extending *FastEvol*, which also deletes the causes of $Employee(jane)$:

DELETE $\{Employee(jane), worksFor(jane, finance)\}$,

hence role-constraining formulas are not derived anymore.

Relation to our approach: Brave semantics $\mathbf{Sem}^{mat}_{brave}$ discussed in *DL-Lite*$_{\text{RDFS}_\neg}$ is inspired by *FastEvol*, which is lifted to deal with DEL/INS/WHERE in SPARQL. This approach though, considers a more expressive ontology language (*DL-Lite*$_{\mathcal{FR}}$), as noted by *CarefulEvol* dealing with side-effects introduced by *FastEvol*; such formulas are not applicable in our setting. We could, in similar way extend to the more expressive *DL-Lite*$_{\mathcal{FR}}$ by taking into account the peculiarities elaborated in this approach.

Integrity constraints: No integrity constraints at the sources are supported.
Mappings language: No mappings are supported.
Ontology language: *DL-Lite*$_{\mathcal{FR}}$.
Update language: INS of atomic assertions (DEL can be implicitly supported).

Propagation/Filtration algorithm [HD92]

Propagate/Filter is one of the most-used algorithms for view maintenance together with *DRed* (see Sec. 2.3). *Propagate/Filter* compared to *DRed*, does not compute the whole overestimation, but rather in each derivation step, for each tuple of the over-estimation checks whether there exists an alternative derivation for the tuple via a query. If there exists an alternative derivation (in this case query would return true), then the tuple is removed from the over-estimation, i.e., it is filtered out. *Propagate/Filter* is used in the bottom-up fashion, where updates are done in the underlying relational sources, expressed in the form of extensional predicates, and the changes should be propagated to intensional predicates which are materialised (stored). The approach works also for both recursive views as well as views with stratified negation. The approach handles negation in a non-monotonic way, for insertion (resp. deletion) of tuples which have negation operator in the body of the rule, are triggered as deletion (resp. insertion) of tuples in the view.

Example 65 (Incremental update using *Propagate/Filter*) Given the OBDM system $\langle \mathcal{O}_m, \mathcal{D} \rangle$, $\mathcal{O}_m = \langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$, $\mathcal{A} = \mathcal{M}(\mathcal{D})$ and TBox $\mathcal{T}$ as defined in Ex. 3; consider the update u:

```
DELETE { :john a :Employee . }
INSERT { :john a :Manager . }
```

Propagate/Filter approach does the following: first, it computes the over-estimation step-by-step, i.e., for the deletion `:john a :Employee`, we apply the axiom `:Employee sc :Person` and obtain the triples: `:john a :Employee.:john a :Person .`. Then for each triple in the current over-estimation, i.e., `:john a :Person` checks whether there exists an alternative derivation for it. The query with rule head `:john a :Person` returns true (is derived from `:john :worksFor :finance`), thus is removed from the over-estimation. Finally, the computed update is:

```
DELETE { :john a :Employee . }
INSERT { :john a :Manager . :john a :Person . }
```

■

Relation to our approach: In this dissertation we have discussed $\mathbf{Sem}_{1a}^{mat}$ and $\mathbf{Sem}_{1b}^{mat}$ which are inspired by DRed. We could in a similar way, for these two semantics create variants based upon *Propagate/Filter* and evaluate their respective performance.

Integrity constraints: No integrity constraints at the sources are supported.
Mappings language: Rules (query over relational sources mapped to a relational view).
Ontology language: Rules (recursive view).
Update language: DEL, INS, DEL-INS of atomic tuples are supported.

Reasoning as Axioms Change [KBB11]

The approach introduces an optimisation step in *DRed* (see Sec. 2.3) in computing the "dependent" facts in the step of overestimation. Dependent facts are referred to all the facts which are implicitly derived from the base facts that are to be deleted. Different from the conventional *DRed* algorithm, which uses semi-naive forward chaining in steps 1 and 3 of delete and rederive, here it exploits fully the materialisation in computing both steps.

Example 66 (Incremental update using optimised DRed) Given the OBDM system $\langle \mathcal{O}_m, \mathcal{D} \rangle$, $\mathcal{O}_m = \langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$, with TBox $\mathcal{T} = \{Employee \sqcap PromotedEmployee \sqsubseteq HappyEmployee$, $Manager \sqsubseteq PromotedEmployee$, $\exists worksFor \sqsubseteq Person$, $HappyEmployee \sqsubseteq Person\}$, ABox $\mathcal{A} = \mathcal{M}(\mathcal{D}) = \{Employee(john), Manager(john)$, $worksFor(john, finance)\}$, and the following update:

```
DELETE { Employee(john) }
```

In Step 1, optimised *DRed* deletes using as input the fixpoint (materialisation):

```
DELETE { Employee(john), HappyEmployee(john), Person(john) }
```

In Step 2, optimised *DRed* re-derives (using as input the fixpoint):

```
INSERT { Person(john) }
```

Note the difference in Step 1 w.r.t. conventional *DRed*, the fixpoint has already the assertion $PromotedEmployee(john)$ which is used to derive $HappyEmployee(john)$, thus Step 1 is computed in just two computation steps. The conventional semi-naive evaluation would need three steps instead of two in this case. For Step 2, it also exploits fix-point, whereas DRed uses the original program rules by performing the computation in semi-naive fashion from scratch. ■

Relation to our approach: For both semantics inspired by DRed, namely $\mathbf{Sem}_{1a}^{mat}$ and $\mathbf{Sem}_{1b}^{mat}$ we could exploit materialisation as described in this approach in order to get better performance results.

Integrity constraints: No integrity constraints at the sources are supported.
Mappings language: No mappings are supported.
Ontology language: Rules.
Update language: DEL of atomic assertions.

Updating Relational Data via SPARQL/Update [HRG10]

OntoAccess is an approach that deals with the problem of updating relational data via SPARQL/UPDATE by resolving mappings in R3M language, which is encoded in RDF and resembles to D2RQ[2]. The mapping language essentially maps database tables to ontology classes and attribute and link tables (i.e., tables in many-to-many relationship) are mapped to properties. Thus, as the result of the mapping, a database row is represented as a set of RDF triples – which is very common. R3M is able to encode integrity constraints as well (primary keys, foreign keys, `NOT NULL`, `DEFAULT`), and thus can check for update violation before the update is propagated to the underlying relational data. The mapping from relational data to RDF is one-to-one, i.e., there is direct relationship between a triple and a table, thus eliminating the possible occurrence of side-effects anomalies common in updates over views. Regarding the translation of SPARQL updates consisted of atomic triples, it is done in 6 steps (*Algorithm 1*, [HRG10]):

- The RDF triples are grouped according to subject, as they represent the same entity,
- Table is identified via URI of subject of the triple and the definition of the mapping,
- Validity of the update is checked whether it violates integrity constraints w.r.t. mapping definition,
- The respective SQL statements are generated using the mapping definition,
- Statements are sorted according to the foreign keys,

[2] `http://d2rq.org`

```
map:empdept a r3m:LinkTableMap ;
  r3m:hasTableName "empdept" ;
  r3m:mapsToObjectProperty :worksFor ;
  r3m:hasSubjectAttribute map:ed_employee ;
  r3m:hasObjectAttribute map:ed_department .

map:ed_employee a r3m:AttributeMap ;
  r3m:hasAttributeName "EMPID" ;
  r3m:hasConstraint [ a r3m:ForeignKey ;
  r3m:references map:employee . ] .

map:employee a r3m:TableMap ;
  r3m:hasTableName "employee" ;
  r3m:mapsToClass :Employee ;
  r3m:uriPattern "employee%%id%%" ;
  r3m:hasAttribute map:employee_id ,
                   map:employee_name ,
                   map:employee_salary .
                  ...
```

Figure 81: The direct mapping in R3M of the link table *empdept* to the object property `:worksFor`, and table *employee* to the class `:Employee` and to the properties `:employee_id, :employee_name, :employee_salary`.

- SQL Statements are executed.

In the case where there is a WHERE clause, first the variables are instantiated , and then the update is done by first deleting atomic data and afterwards inserting atomic data. In cases where update is consisted of both DELETE and INSERT, each of the previous 6 steps are used, and the outcome of the translation is similar.

Example 67 (Updating relational data via SPARQL/Update) Given the OBDM system $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$, where schema $\mathcal{S}$ and instances $\mathcal{D}$ defined as in Ex. 1, the mappings $\mathcal{M}$ are defined in R3M in Ex. 67. As this approach does not take into account the ontology, thus we have $\mathcal{T} = \emptyset$. Consider the update:

```
DELETE { :emp1 :worksFor :dept102 }
INSERT { :dept2 a :Department }
```

Translation takes into account the database instance $\mathcal{D}$ and accordingly translates it to either INSERT or UPDATE:

```
DELETE FROM empdept
WHERE EMPID=1 AND DEPTID=102;

UPDATE department
SET DeptName = NULL
WHERE ID=102;
```

In this particular case, for DELETE template, the set of triples coincide with the number of attributes in the underlying table, hence the translation is done using the same

```
map:employees a d2rq:ClassMap;
  d2rq:dataStorage map:database;
  d2rq:uriPattern "emp@@employee.ID@@";
  d2rq:class :Employee.

map:empName a d2rq:PropertyBridge;
  d2rq:belongsToClassMap map:employees;
  d2rq:property :name;
  d2rq:column "employee.NAME".

map:salary a d2rq:PropertyBridge;
  d2rq:belongsToClassMap map:employees;
  d2rq:property :salary;
  d2rq:column "employee.SALARY".
                ...
```

Figure 82: The mapping of table *employee* as instances of class `:Employee`; mapping attributes to properties `:name` and `salary` respectively.

(DELETE) operator; whereas INSERT is translated to an UPDATE, since already exists a tuple with ID equal to 102. ■

Relation to our approach: The approach is similar to updates in DBpedia discussed in this dissertation, in the sense that it computes atomic updates from a general SPARQL update, and then groups them based on subject before resolving w.r.t. mappings. This approach encodes integrity constraints directly in the mapping definition; in similar way we could enrich DBpedia mappings in order to statically check updates without propagating them to the sources.

Integrity constraints: Primary keys, foreign keys, `NOT NULL`, `DEFAULT`.
Mappings language: Mappings language called R3M.
Ontology language: TBox axioms are not supported.
Update language: General DEL/INS/DEL-INS updates via SPARQL/Update.

Updating relational data via virtual rdf [EK12]

D2RQ/Update is an extension to D2RQ to support updates posed over an RDF view. Mapping from relational data to RDF vocabulary is done via D2RQ—a known mapping language for lifting relational data—which can be done either manually or automatically. D2RQ/Update strives to minimize the number of generated SQL statements, while bypassing the integrity constraints. The approach here sorts the triples by the subject, then it does a topological sort of the triples to be added, based on the order they were added. Regarding the integrity constraints, besides the primary/foreign keys, it emphasises on `NOT NULL` constraints on attributes.

Example 68 (Updating relational data using D2RQ/Update) Given an OBDM system $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$, where schema $\mathcal{S}$ and instances $\mathcal{D}$ defined as in Ex. 1. In addition,

we explicitly set attribute SALARY to `NOT NULL`. The mappings $\mathcal{M}$ are defined in D2RQ (cf. Fig. 82). Consider the following update u:

```
INSERT { :emp3 :worksFor :dept103 . :emp3 :name "Joe" .
                                    :emp3 :salary "2000"^xsd:integer }
```

Note that from the subject :emp3 in `:emp3 :name "Joe"`, the mapping is able to resolve that it should be propagated into *employee* table without a need for a triple of type `:emp3 a :Employee`. The approach performs the translation based on the set of triples, optimising the number of SQL statements and re-ordering them in order to preserve the integrity constraints:

```
INSERT INTO employee VALUES(3, "Joe", 2000);
INSERT INTO department VALUES(103, NULL);
INSERT INTO empdept VALUES(3, 103);
```

■

Relation to our approach: In our case of DBpedia updates, we could potentially re-use the idea behind this approach, in order to check for source constraints based on the history they were introduced, e.g., checking infobox properties based on the time they were added/updated by exploiting the Wikipedia history. Based on that, we could optimize and re-order the translated updates.

Integrity constraints: A handful of integrity constraints at the sources are supported.
Mappings language: D2RQ.
Ontology language: TBox axioms are not supported.
Update language: General DEL/INS/DEL-INS updates via SPARQL/Update.

Bi-directional translation of relational data into virtual RDF stores [RKK+10]

The approach named D2RQ++ uses the mapping language D2RQ, while also introducing algorithms for dealing with updates over blank nodes in RDF view. It extends D2RQ mapping language with three type of constructs for blank nodes, which are used for capturing structures from a single table (cf. Ex. 69), tables in one-to-many relationships and many-to-many relationships respectively. When posing updates, D2RQ++ for all triples that can not be propagated—as they would violate the integrity constraints or have no entity matching—it hosts them in a separate triple store. The triples in triple store will be checked against the database on regular intervals, i.e., by consolidating with the database and performing the updates as necessary. In case a certain attribute of a tuple is updated with a NULL value, then the corresponding triple in the triple store would be deleted as well. It is worth mentioning that the SPARQL SELECT queries are posed against the RDF view and triple store altogether. The main criticism of this approach is that there is inconsistency of the RDF view w.r.t. the legacy applications that work with relational data, due to triples hosted in the triple store.

```
map:employee_address a
  d2rqrw:SimpleLiteralBlankNodePropertyBridge;
  d2rq:belongsToClassMap map:employee;
  d2rq:property vocab:employee_address;
  d2rq:pattern "@@employee.address_street@@/
@@employee.address_city@@/
@@employee.address_state@@".

map:employee_address_street a d2rq:PropertyBridge;
  d2rqrw:belongsToBlankNode map:employee_address;
  d2rq:belongsToClassMap map:employee;
  d2rq:property vocab:employee_address_street;
  d2rq:column "employee.STREET" .

map:employee_address_city a d2rq:PropertyBridge;
  d2rqrw:belongsToBlankNode map:employee_address;
  d2rq:belongsToClassMap map:employee;
  d2rq:property vocab:employee_address_city;
  d2rq:column "employee.CITY" .

map:employee_address_state a d2rq:PropertyBridge;
  d2rqrw:belongsToBlankNode map:employee_address;
  d2rq:belongsToClassMap map:employee;
  d2rq:property vocab:employee_address_state;
  d2rq:column "employee.STATE" .
                ...
```

Figure 83: Mapping *employee* table to property `:employee_address`, which has a blank node as a property value and has the following three properties `address_street, address_city, address_state`.

Example 69 (Bi-directionality using D2RQ++) Given an OBDM system $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$, where schema $\mathcal{S}$ defined as:

employee[ID:**INTEGER**, STREET:**STRING**, CITY:**STRING**, STATE:**STRING**],

and no instances, i.e., $\mathcal{D} = \emptyset$. As one can use a blank node to represent the mapped data from attributes STREET, CITY and STATE. The mappings $\mathcal{M}$ in D2RQ using extended constructs for representing blank nodes are defined as in Fig. 83. Consider the following update u_1:

```
INSERT { :emp1 :employee_address [
                :employee_address_street "Welthandelsplatz" .
                :employee_address_city "Vienna" .
                :employee_address_state "AUT" ] . }
```

which would be propagated as:

```
INSERT INTO employee
VALUES(1, "Welthandelsplatz");

UPDATE employee
```

```
SET CITY="Vienna"
WHERE ID=1;

UPDATE employee
SET STATE="AUT"
WHERE ID=1;
```

Consider we have another update u_2:

```
INSERT { :emp1 :employee_address
               [ :employee_address_street "Favoritenstrase" ] . }
```

given that the attribute ID is the key and the corresponding attribute value for table employee is not NULL (for ID=1), then the triple would be inserted in the triple store. Then, given q (note $\mathcal{T} = \emptyset$) as follows:

```
SELECT ?X
WHERE
{
        :emp1 :employee_address [ :employee_address_street ?X ] .
}
```

would return both $\mu(?X)$="Welthandelsplatz" and $\mu(?X)$="Favoritenstrasse". ■

When inserting triples that constitute a blank node as shown in the previous example, then they would be inserted only if all the triples do not exist as tuples in the underlying database, otherwise if at least one corresponding tuple exists, then all the triples would be stored in the triple store. This behaviour can be changed if desired to other specific needs though. In case of updating link tables, i.e., *empdept* then it would only update the link table if there exist reference rows in the respective tables, i.e., *employee* and *department*. Otherwise, the triple would be inserted in the triple store.

Relation to our approach: The approach discussed in this work is similar to resolving updates in DBpedia that have mappings of type *IntermediateNodeMappings*. We could also relax our approach and allow more values for a property by allowing redundant triples to be stored in a dedicated triple store.

Integrity constraints: A handful of integrity constraints at the sources are supported.
Mappings language: D2RQ.
Ontology language: TBox axioms are not supported.
Update language: General DEL/INS/DEL-INS updates via SPARQL/Update.

RESTful Writable APIs for the Web of Linked Data using Relational Storage Solutions [HG11]

In this approach, relational data are mapped to RDF using R2RML mapping language. The mapping is restricted such that the table must be unique for every triple pattern in a SPARQL SELECT query (WHERE clause), which avoids ambiguities in update

```
_:mapping1 rr:table "employee" ;
           rr:subjectMap [rr:column "ID"] ;
           rr:propertyObjectMap [ rr:property <test:name> ;
                                  rr:column "NAME" ;
                                  rr:columnGraphIRI <test> ] ;
           rr:propertyObjectMap [ rr:property <test:salary> ;
                                  rr:column "SALARY" ;
                                  rr:columnGraphIRI <test> ] .
_:mapping2 rr:table "triplestore" ;
           rr:subjectMap [ rr:column "S" ] ;
           rr:propertyObjectMap [ rr:propertyColumn "P" ;
                                  rr:column "O" ;
                                  rr:columnGraphIRI <test> ] .
                ...
```

Figure 84: Mapping 1 specifies the mapping of *employee* table to quads with context $\langle test \rangle$ having properties name and salary and subject value the "ID" attribute. Mapping 2 specifies a generic mapping where "S" is the subject of the quad with context $\langle test \rangle$, whereas attribute "P" specifies the property name having property value the attribute "O".

translations. The advantage of using R2RML mapping language is on creating any kind of RDF triples including blank nodes, reification and NAMED graphs (quads) mapped from the relational data. The authors introduce algorithms for SELECT queries, as well as DELETE and INSERT for SPARQL/Update, and their translations to SQL queries and SQL DML respectively. The general updates are transformed to atomic updates by instantiating the variables from WHERE clause, and then performing the translation. As for an INSERT operation there can be several translations, it uses a metric that counts for the number of attributes and rows inserted, thus always choosing a translation that has the least data operations.

Example 70 (Updating relational data using SPARQL/Update) Given the OBDM system $\langle \mathcal{O}_m, \mathcal{D} \rangle$, $\mathcal{O}_m = \langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$, where *employee* schema as in Ex. 1 together with given another table schema defined as:

$$triplestore[\texttt{S}\texttt{:STRING}, \texttt{P}\texttt{:STRING}, \texttt{O}\texttt{:STRING}]$$

Let $\mathcal{D}$={employee(1, NULL, 1400)}. The mappings $\mathcal{M}$ in R2RML are defined as in Fig. 84. Given the following update u_1:

```
INSERT { :emp1 :name "bob" . }
```

given that there is a corresponding value equal to NULL in the *employee* table, according to *Algorithm 4* [HG11] would be translated to:

```
UPDATE employee
SET NAME = "bob"
WHERE ID=1;
```

Note that this has less data operations than inserting a new tuple in the other mapped table *triplestore*. Now, let us consider another update u_2 after u_1:

```
INSERT { :emp1 :name "bobby" }
```

then it will be inserted in the table *triplestore* (this second update with the same property name will not be issued in the *employee* table because we would overwrite the previous name):

```
INSERT INTO triplestore(S, P, O)
VALUES (1, "name", "bobby") }
```

Then, given a query q_1 (note $\mathcal{T} = \emptyset$) as follows:

```
SELECT ?X
WHERE { :emp1 :name ?X }
```

according to *Algorithm 3* [HG11], both $\mu(?X)$="bob" and $\mu(?X)$="bobby" would be returned fetched from both tables. Lastly, if we are given the following update u_3:

```
DELETE { :emp1 :name "bob", :emp1 :salary "1400"^^xsd:integer }
```

according to *Algorithm 6* [HG11] would be translated to:

```
UPDATE employee
SET NAME=NULL, SALARY=NULL
WHERE ID=1;
```

Since the tuple with ID=1 has all columns with NULL values, then it would be removed in the next step by another consecutive update:

```
DELETE FROM employee
WHERE ID=1;
```

■

As can be seen from Ex. 70, INSERT queries are translated to either UPDATE or INSERT, whereas DELETE queries are always translated to UPDATE queries, thus updating the respective attribute values with NULLs. A tuple would only be removed if it has NULL values for all of its attributes.

Relation to our approach: This approach allows for having multiple values for a certain property by having a separate table which stores those tuples. In the setting of DBpedia updates, we could allow for multiple property values by either storing them in a relational table, or in a triple store as RDF triples (as it is the case with the previous approach [RKK+10]).

Integrity constraints: A handful of integrity constraints at the sources are supported.
Mappings language: R2RML.

Ontology language: TBox axioms are not supported.
Update language: General DEL/INS/DEL-INS updates via SPARQL/Update.

Update Semantics of Relational Views [BS81]

This seminal work is considered to be one of the most important and provided new understanding to the theory of view updating. In this context, view definitions are considered as functions f, instances as states s, and updates as mappings transitions $u : s \to s'$. The basic idea is that when updating a view, a new view definition g is considered (so called "view complement"), which together with the initial view definition f, namely the composition $f \times g$ [Ber03] gives the complete state of the database $\mathcal{D}$ in terms of tuples $(f(s), g(s))$, such that there is a one-to-one correspondence between database states s and the pairs $(f(s), g(s))$. Once an update is posed over the view, it has the complete state of the underlying database (the "view complement" would add all the hidden information from the database, and would distinguish between the states that map onto the same view state under f) and thus unambiguously can translate the updates. The composition of these two view definitions would yield an injective view definition, which has as inverse the function $(f \times g)^{-1}$. In the paper are considered two criteria for updates which should be satisfied, i.e. the translation of updates should be: *consistent* (the update should be translated so that it exactly affects the view) and *acceptable* (the update should not do anything to the sources, if the update would not do anything to the view). The drawbacks of this approach primarily are first, due to the view complement—even though can be always computed—is not unique in general (in this case it is up to the database administrator to choose one based on application requirements), and second, an update should not change the state of the view complement (i.e. should not change the state of the database on which the view complement relies). The latter restriction is imposed because an update translation should always yield a unique translation for each state of the database (the state of the database does not play a role in an update translation). As a last remark, in order that an update u be translatable under g (g-translatable), then the tuple should lie in the image of the database state under $f \times g$, in that case it represents a unique database state s', which is the outcome of the update.

Example 71 Given the OBDM system $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$, where we have the following schema:

employees[Employee:**STRING**, Department:**STRING**, Manager:**STRING**],

the database instance $\mathcal{D}$ consists of a single tuple:

employees={(John, Finance, Jack)},

whereas mappings $\mathcal{M}$ are defined as follows:

m_1 : **SELECT** Employee, Department **FROM** Employee $\rightarrow$ $worksFor(\mathbf{emp}(Employee), \mathbf{emp}(Department))$

m_2 : **SELECT** Department, Manager **FROM** Employee $\rightarrow$ $manager(\mathbf{emp}(Department), \mathbf{emp}(Manager))$

Note that the composition of the mappings $m_1 \times m_2$ yields an injective mapping. In this case we fix m_2 as the "view complement", and we assume it remains constant (unchanged) after the update. Consider that we have the following update u:

```
INSERT { worksFor(Jane, Finance) }
```

If we are given only the mapping m_1 then we cannot translate u, but if are given the view complement m_2, then we can uniquely translate this update to:

```
INSERT { employees("Jane", "Finance", "Jack") }
```

In other words, the assertion $manager(Finance, Jack)$ is needed in order the update u to be g-translatable, since there exists a new updated state in the database (i.e., the one after the translated update is committed):

employees={(John, Finance, Jack), (Jane, Finance, Jack)}

corresponding to the pair $(uf(s), g(s))$ in the view under $f \times g$:

$worksFor(Jane, Finance), worksFor(John, Finance), manager(Finance, Jack)$

■

Relation to our approach: This approach uses joins of mappings in order to translate an update. DBpedia mappings currently do not have the expressivity of having joins, but in case they support such constructs in the future, the approach of computing a "view complement" would be an option to translate the updates.

Integrity constraints: A handful of integrity constraints at the sources are supported.
Mappings language: Mappings as generic functions.
Ontology language: TBox axioms are not supported.
Update language: Atomic DEL/INS/UPD.

On the evolution of the instance level of DL-Lite knowledge bases [LS11]

The approach proposes tractable algorithms for deleting and inserting ABox-es over DL-Lite$_{\mathcal{A},id}$ ontologies, which is one of the most expressive fragments of DL-Lite family of logics. This fragment contains, among others[3], identification assertions (IDs) (cf. [CDGL$^+$08]). Deletions are easier than inserts, as the set of atoms and their causes are computed. Inserts are based upon *WIDTIO* (see Sec. 2.3), which is considered quite radical given that as result yields the atoms that hold in every possible consistent knowledge base. Nevertheless, the advantage of WIDTIO is that the result after an update

[3]DL-Lite$_{\mathcal{A},id}$ is one of the most expressive DLs in the DL-Lite family, because this family includes also DLs with n-ary relationships, DLs allowing for conjunction in the left-hand side of inclusions, DLs combining the above features, and also DLs with general forms of denials (not only disjointness between concepts and roles).

is always expressible in the same logic. Given that the computation of all such possible consistent knowledge bases is exponential in the worst case in this particular logic, it takes a different approach for computing the same result: for each ground atom in the materialisation $mat(\mathcal{A} \cup \mathcal{T})$ minus the ground atoms to be inserted in u, it checks whether for such atom there exists a $\mathcal{T}$-violation set—a minimal set in which such atom contributes to a clash—in $mat(\mathcal{A} \cup \mathcal{T}) \cup mat(u \cup \mathcal{T})$. If such atoms participate in the $\mathcal{T}$-violation set, then they are consequently removed from the knowledge base.

Example 72 Given the OBDM system $\langle \mathcal{O}_m, \mathcal{D} \rangle$, $\mathcal{O}_m = \langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$, where TBox $\mathcal{T} = \{\exists worksFor^- \sqsubseteq Department,\ id \quad Manager \quad worksFor, Manager \sqsubseteq \neg Employee\}$ ABox $\mathcal{A} = \mathcal{M}(\mathcal{D}) = \{worksFor(jane, finance),\ Manager(jane),\ Employee(john)\}$ and the update:

$u =$ **INSERT** $\{Manager(john), worksFor(john, finance)\}$.

According to *WIDTIO* semantics one would get two consistent ABox-es:

$$\mathcal{A}_1 = \{worksFor(jane, finance), Manager(john), worksFor(john, finance), Department(finance)\}$$

$$\mathcal{A}_2 = \{Manager(jane), Manager(john), worksFor(john, finance), Department(finance)\}.$$

Their intersection would yield the final result:

$$\mathcal{A}' = \mathcal{A}_1 \cap \mathcal{A}_2 = \{Manager(john), worksFor(john, finance), Department(finance)\}.$$

The *ComputeInsertion* algorithm (Algorithm 2, [LS11]) efficiently computes the updates described before. The fact $worksFor(jane, finance)$ together with $Manager(jane)$ both contribute to $\mathcal{T}$-violation set w.r.t. the update u, by violating the identity constraint; as such they are both removed. Also, one can see that $Employee(john)$ itself is in clash with the update $Manager(john)$; as such it is removed. The result obtained in this way is the same as computing the intersection as before.

Relation to our approach: The approach uses *WIDTIO* by inserting the update on hand and overriding the conflicting knowledge in the setting of the most expressive DL-Lite family of logics. This method could be potentially used for extending $\mathbf{Sem}^{mat}_{brave}$ to this fragment of logics.

Integrity constraints: No integrity constraints at the sources are supported.
Mappings language: No mappings are supported.
Ontology language: *DL-Lite*$_{\mathcal{A},id}$.
Update language: Atomic DEL/INS.

Practical Update Management in Ontology-Based Data Access [DLO$^+$17]

The approach handles both source-level and ontology-level updates in the context of *DL-Lite*$_\mathcal{A}$ OBDA. The updates on the source-level via mappings are transformed to ontology-level updates, which as a result can be intrinsically inconsistent (cf. Ex. 22). The approach is not a classic view update problem, in the sense that it does not propagate updates directly to the original data at the source level, but it uses a new dedicated database to store such changes. These auxiliary tables are specially tailored for both deletes and inserts. In a nutshell, the update semantics does the follow: for deletions, it deletes the facts plus their causes; for inserts, it inserts the facts and their effects by overriding the contradicting facts and their causes. The computation of updates is done using a non-recursive Datalog program $Datalog(\mathcal{T}, \mathcal{M})$—based on TBox $\mathcal{T}$ and mappings $\mathcal{M}$—as described in [DORS16]. The base facts are the facts contained in the database plus the update. Let us illustrate it via an example, where we focus on ontology-level updates only.

Example 73 Consider an OBDM system $\langle \mathcal{O}_m, \mathcal{D} \rangle$, $\mathcal{O}_m = \langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$, where $\mathcal{S} = \{employees, managers\}$, $\mathcal{A} = \{Employee(john)\}$, $\mathcal{T} = \{Manager \sqsubseteq Person, Employee \sqsubseteq Person, Manager \sqsubseteq \neg Employee\}$, $\mathcal{M} = \{m_1, m_2\}$, such that:

m_1: `Employee(x) :- employees(x)`
m_2: `Manager(x) :- managers(x)`

According to $Datalog(\mathcal{M}, \mathcal{T})$ we get the following rules:

```
1) del_Manager'(x)  :- Manager(x), del_Person_ol(x)
2) del_Employee'(x) :- Employee(x), del_Person_ol(x)
3) del_Manager'(x)  :- Manager(x), ins_Employee_ol(x)
4) del_Employee'(x) :- Employee(x), ins_Manager_ol(x)
5) ins_Person'(x)   :- del_Manager'(x), ¬del_Person_ol(x)
6) ins_Person'(x)   :- del_Employee'(x), ¬del_Person_ol(x)
```

Rules 1),2) are generated to take into account deletion of causes; rules 3),4) to override facts that are inconsistent w.r.t. new facts; and rules 5),6) are to keep the common logical consequences (dangling effects) in case of deletion of a fact. Consider the following update:

```
DELETE { Person(john) }
INSERT { Manager(joe) }
```

We have the base facts from the update $del\_Person\_ol(john), ins\_Manager\_ol(joe)$ and from the database $Employee(john)$.

From the rules we compute deletions and insertions:

$del\_Employee'(john), ins\_Manager(joe)$.

In other words, we are performing the update:

```
DELETE { Employee(john) }
INSERT { Manager(joe) }
```

Note that deletion of $Person(john)$ is not necessary, given that we are in a virtual OBDM setting and thus deleting $Employee(john)$ suffices. Likewise, inserting $Person(joe)$ is not needed, given that it is derived from the TBox.

In the end, the original mapping is transformed in the so-called *write-also* mapping:

```
Employee(x)   :-  employees(x), ¬del_Employee(x)
Employee(x)   :-  ins_Employee(x)
Manager(x)    :-  managers(x), ¬del_Manager(x)
Manager(x)    :-  ins_Manager(x)
```

■

Relation to our approach: This approach is essentially $\mathbf{Sem}^{mat}_{brave}$ lifted in the context of (virtual) OBDM, as such the Datalog program consisting of TBox and mappings could be implemented in the context of SPARQL as well [Pol07]. Also, herein are discussed intrinsically inconsistent updates caused by source-level updates, in the same spirit as safe rewriting discussed in this dissertation.

Integrity constraints: A handful of integrity constraints at the sources are supported.
Mappings language: Rules.
Ontology language: *DL-Lite*$_{\mathcal{A}}$.
Update language: Atomic DEL/INS/DEL-INS.

8.1.2 TBox Updates in Ontology-Based Data Management

Updating RDFS: from theory to practice [GHV11]

The approach concerns with deleting TBox-es in the RDFS context. The algorithm for deleting schema triples, so-called delta candidates (i.e., minimal schema triples to be deleted), is based upon performing a minimal multicut on the graph on both sc and sp axioms separately. When performing a multicut, there is non-deterministic choice of which axiom is selected for deletion, as they are equally possible thus being exponential in the worst case. But for small schemas number of such choices is small, thus being more feasible. Notably, deleting triples of type dom and rng is done by just simply deleting the respective axiom. Deleting a graph boils down to deleting individual triples separately, and creating a set from the union of the resulting triples.

Example 74 (Deleting TBox axioms) Given the OBDM system $\langle \mathcal{O}_m, \mathcal{D} \rangle$, $\mathcal{O}_m = \langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$, with components as defined in Ex. 1 bravely merged with $\{Manager \sqsubseteq$

Employee} (i.e., the conflicting knowledge in TBox is removed), and an ontological axiom to be removed u_1 = **DELETE** {$Manager \sqsubseteq Person$}. According to (*Algorithm1*, [GHV11]), the materialisation of the graph is computed and then the graphs as in the Fig. 85 are created. Then, a minimal multicut on the computed graphs G[sp] and G[sc] is performed. In our example, the multicut would be performed on the graph G[sc] where either pairs of triples $\{(n_{Manager}, n_{Employee}), (n_{Manager}, n_{Person})\}$, or pairs of triples $\{(n_{Employee}, n_{Person}), (n_{Manager}, n_{Person})\}$ are chosen to be removed, i.e., deletion is non-deterministic:

```
DELETE { :Manager sc :Employee . :Manager sc :Person }
```

or

```
DELETE { :Employee sc :Person . :Manager sc :Person }
```

Now, suppose we are given the state before the u_1, and given another update u_2:

```
DELETE { :worksFor dom :Person }
```

then again using the same algorithm the minimal multicut would be on both G[sp] and G[sc], thus we get:

```
DELETE { :worksFor dom :Person . :Employee sc :Person }
```

■

Relation to our approach: This approach performs a minimal multicut on a graph in order to deal with deletion of implicit TBox axioms. This deletion as also exemplified above has non-deterministic result. In this dissertation we have proposed $\mathbf{Sem}^{mat}_{incut}$ and $\mathbf{Sem}^{mat}_{outcut}$ which both have deterministic results and can be rewritten using SPARQL property paths, and both achieve the same result as doing a minimal multicut on a graph (cf. Prop. 7).

Integrity constraints: No integrity constraints at the sources are supported.
Mappings language: No mappings are supported.
Ontology language: Minimal RDFS - sc, sp, dom, rng (see Fig. 23).
Update language: DEL of atomic triples in a graph (INS can be implicitly supported).

TBox Evolution in DL-Lite [CKNZ10]

The approach deals with TBox evolution in DL-Lite knowledge bases. For TBox evolution same as with ABox evolution (cf. Sec. 8.1.1) the same postulates should be satisfied, i.e. *Coherence*[4], *Minimality of change* and *Protection*[5]. The state-of-art formula-based

[4]Coherence is a stronger requirement than consistency because not only the knowledge base should be satisfiable, but also each concept from vocabulary must have at least one individual assigned in ABox.

[5]That is, when a portion of TBox (or ABox) is declared as protected, it means that it cannot be subject of any TBox (or ABox resp.) changes and thus it remains always static after updates.

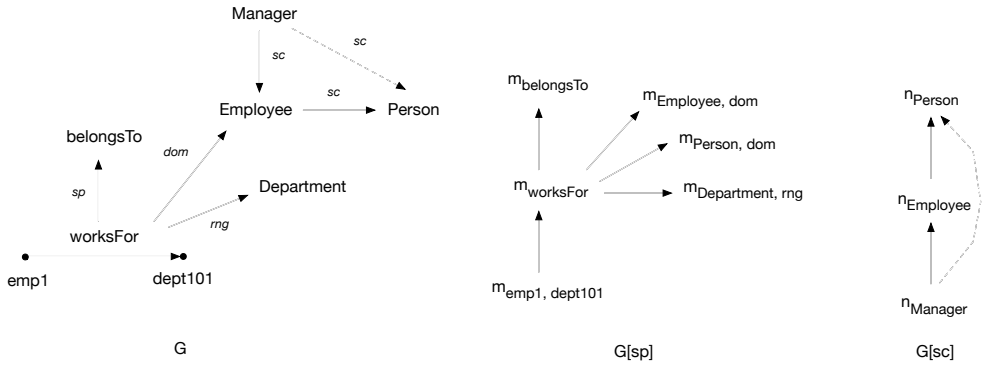


Figure 85: The directed graph is generated given the ontology in Ex. 74. The dashed lines represent implicit triples as computed by the materialisation. From the graph G are distinguished both sp and sc axioms and denoted using the respective G[sp] and G[sc] graphs.

semantics have the caveat of inexpressibility (because of lack of disjunction in DL-Lite), but the paper proposes a pragmatic variant of it called "bold semantics", which given a $\mathcal{O}$ and $\mathcal{U}$ always yields a maximal non-deterministic result. The advantage of this approach is that it always computes the result in polynomial time, whereas the disadvantage is the non-determinism. The algorithm starts from the empty set of TBox axioms and then it enumerates over all individual TBox axioms non-deterministically (without any imposed order), by adding them and making a check whether the ontology is coherent. If that is the case, then they are added to the set returned TBox axioms and this process is continued iteratively.

Example 75 Given the OBDM system $\langle \mathcal{O}_m, \mathcal{D} \rangle$, $\mathcal{O}_m = \langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$, with components as defined in Ex. 1 merged with $TopManager \sqsubseteq Manager$, and TBox to be added $u =$ **INSERT** $\{TopManager \sqsubseteq \neg Person\}$. According to the "bold semantics", the result is non-deterministic, i.e., the result is:

```
DELETE { :TopManager sc :Manager }
```

or

```
DELETE { :Manager sc :Person }
```

posed over $\mathcal{O}_m$.

■

Relation to our approach: The bold semantics discussed in this approach could be used to extend the semantics discussed in this dissertation, namely $\mathbf{Sem}^{mat}_{incut}$ and $\mathbf{Sem}^{mat}_{outcut}$, and lift them from $DL\text{-}Lite_{\text{RDFS}}$ into the setting of more expressive logics $DL\text{-}Lite_{\mathcal{FR}}$.

Integrity constraints: No integrity constraints at the sources are supported.
Mappings language: No mappings are supported.
Ontology language: *DL-Lite*$_{\mathcal{FR}}$.
Update language: Atomic INS (DEL can be implicitly supported as well).

Reasoning as Axioms Change [KBB11]

The approach also supports the updates over rules instead of just facts, the latter as elaborated in Sec. 8.1.1. Deleting rules is performed by merely deleting the designated explicit rule, and then performing the DRed if update contains both rule and atomic atoms. Though it is not possible to delete any implicit rule.

Example 76 Let we interpret the ground atoms as ABox and rules as TBox axioms respectively. Then, given the OBDM system $\langle \mathcal{O}_m, \mathcal{D} \rangle$, $\mathcal{O}_m = \langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle^{\mathbb{S},\mathbb{L}}$, with components as in Ex. 1, and update as follows:

```
DELETE { :john :worksFor :finance . :worksFor rng :Department . }
```

according to the approach it would be translated to (first TBox axioms are deleted):

```
DELETE { :worksFor rng :Department . }
```

afterwards, for ABox-es the same algorithm as described in Sec. 8.1.1, is performed.

Step 1: delete over-estimation:

```
DELETE { :john :worksFor :finance . :john a :Person . }
```

Step 2: re-derive:

```
INSERT { :john a :Person . }
```

Note that `:finance a :Department` is not in the set of ABox axioms to be removed, as `:worksFor rng :Department` is a rule removed previously. Due to TBox changes is not possible to use here fixpoint as input to compute faster the derivations, as it was the case with ABox only updates.

■

Relation to our approach: This is a straightforward method to handle TBox+ABox updates, which can be similarly incorporated in our semantics for handling TBox+ABox updates, e.g., by first applying $\mathbf{Sem}^{mat}_{incut}$ and then followed by $\mathbf{Sem}^{mat}_{2}$.

Integrity constraints: No integrity constraints at the sources are supported.
Mappings language: No mappings are supported.
Ontology language: Rules.
Update language: Atomic DEL.

8.1.3 Discussion of Related Approaches Versus this Dissertation

In Sec. 8.1.1 and Sec. 8.1.2 we have thoroughly discussed the related (state-of-the-art) approaches, which fall into the category of updates over views, view maintenance, updates over DL knowledge bases and updates in the context of OBDM.

From their analysis, we can deduce that—in the most cases—there is an "exclusive or" support for TBox axioms versus mappings, i.e., whenever TBox is supported then mappings are not, and vice versa.

Previous work on updates in the context of tractable ontology languages such as RDFS and *DL-Lite* (cf. Sec. 8.1.1) typically has treated DELETEs and INSERTs in isolation, but not both at the same time nor in combination with templates filled by WHERE clauses, as in SPARQL 1.1; that is, these approaches are not based on BGP matching but rather on a set of ABox assertions to be updated, known a priori. Pairing both DELETE and INSERT, as treated in this dissertation, poses new challenges, which we tried to address here in the practically relevant context of triple stores.

The related approaches and the update semantics discussed therein, often maintain a materialised knowledge base, while none of them treat the other extreme of maintaining a reduced knowledge base. In this dissertation, we discussed in the context of triple stores various update semantics which are materialise-preserving, as well as the ones which are reduce-preserving respectively.

Yet another observation is that approaches that deal with the problem of updates over views used mappings without restrictions, whereas in the context of updates in OBDM, the mappings are mainly of one-to-one type. This pragmatic approach of having mappings of one-to-one type, is somewhat expected given that TBox axioms render the problem harder by adding another layer of complexity.

In the context of updates in OBDM discussed in this dissertation, we proposed update semantics in the DBpedia setting, where not only mappings are more expressive (many-to-many DBpedia mappings), but also the (DBpedia) ontology language is quite expressive in OWL 2 RL.

Finally, as regards to TBox updates, the related approaches (cf. Sec. 8.1.2) typically perform a TBox update with a non-deterministic outcome. In this dissertation, we have opted for two semantics that yield a deterministic result when doing a (TBox) triple deletion, and can easily be re-written using SPARQL 1.1 property paths.

8.2 Conclusions & Future Work

The motivation for this dissertation has been the lack of a standard defining the interplay between SPARQL updates and entailment regimes, which was left open in the resp. SPARQL 1.1 specifications [GPP13, GOH+13], in the case of an ABox managed by a triple store, or also in the more general context of Ontology-Based Data Management (OBDM) involving a relational DBMS where mappings are present.

We have investigated different update semantics, starting with TBox expressivity from minimal RDFS, adding disjointness, and finally adding OWL 2 RL features to capture the DBpedia setting. For the first two cases, we have elaborated on the rationale of the update semantics by checking against a set of postulates, which we defined inspired by AGM postulates.

The choice of the update semantics can be done based on the use cases or/and on the postulates that they fulfill. In general, based on the use cases and the accompanying examples we have elaborated for the semantics, as well as the postulates they fulfill, we conclude that there is no "one-size-fits-all" semantics. This claim is not surprising given that this behavior has always been the case when tackling updates on a knowledge base.

In general, we mention that the update semantics which fulfills the most postulates (12) is $\mathbf{Sem}_{1b}^{mat}$—thanks to distinguishing between explicit and implicit triples—whereas the ones that fulfill the least number of postulates (8) are $\mathbf{Sem}_{3}^{mat}$, $\mathbf{Sem}_{caut}^{mat}$ and $\mathbf{Sem}_{faint}^{mat}$. $\mathbf{Sem}_{3}^{mat}$ deletes too much facts, while $\mathbf{Sem}_{caut}^{mat}$ and $\mathbf{Sem}_{faint}^{mat}$ when dealing with inconsistencies do not give priority to the new facts to be inserted.

While ontologies should be "ready for evolution" [NK04], more work into semantics for updates of ontologies alongside with TBox & ABox data is still needed to ground research in *Ontology Evolution* into Semantic Web standards: SPARQL, RDF, RDFS and OWL. In particular, this is needed in the light of the emerging importance of RDF and SPARQL in domains where data is continuously updated, e.g., dealing with dynamics in Linked Data, querying sensor data, or stream reasoning.

In the area of database theory, there has been a lot of work on updating logical databases: Winslett [Win05] distinguishes between model-based and formula-based updates. The update semantics discussed in this dissertation clearly falls in the latter category; more concretely, ABox updates could be viewed as sets of propositional knowledge base updates [KM89] generated by SPARQL instantiating DELETE/INSERT templates.

In the more applied area of databases, there are obvious parallels between some of our considerations and CASCADE DELETEs in SQL. That is, deletions under foreign key constraints, in the sense that we trigger additional deletions of causes/effects in some of the proposed update semantics discussed herein.

8.2.1 *DL-Lite*$_{\text{rdfs}}$

First, we answered the research questions posed in Sec. 1.3 by proposing different semantics of SPARQL 1.1 Update in the context of interplay with *DL-Lite*$_{\text{RDFS}}$:

> Given a SPARQL update request, how can we preserve materialisation or reducedness of the triple store while capturing the intuition of the update? Which additional triples should be deleted or inserted in order to achieve this intuition?

To the best of our knowledge, this is the first work to discuss how to combine RDFS with the SPARQL 1.1 Update language. While initially we have been operating on a

very restricted setting (only capturing minimal RDFS entailments, restricting BGPs to disallow non-standard use of the RDFS vocabulary), we could demonstrate that even in this setting the definition of a SPARQL 1.1 Update semantics under entailments is a non-trivial task. We proposed several possible semantics in the case of ABox updates for both materialised and reduced triple stores, neither of which might seem intuitive for all possible use cases; this suggests that there is no "one-size-fits-all" update semantics. Likewise, for TBox updates, we proposed two semantics in the context of materialised triple stores, which both yield deterministic results.

Next, we discussed a more in-depth investigation of our postulates in the light of the more general considerations in classical AI on theory change and belief revision [AM82, AGM85]. From the results (cf. Table 41), we see that unless one's strategy is to remove the causes, in which case one would choose $\mathbf{Sem}_2^{mat}$, then the optimal strategy is to choose $\mathbf{Sem}_{1b}^{mat}$. As regards to the subtle differences between $\mathbf{Sem}_{1a}^{mat}$ and $\mathbf{Sem}_{1b}^{mat}$: first, they yield different output as discussed in Ex. 27; second, from the postulates we can see that keeping explicit and implicit triples separate as in $\mathbf{Sem}_{1b}^{mat}$ has the advantage that updates are always idempotent, whereas $\mathbf{Sem}_{1a}^{mat}$ has the advantage that different syntactic updates but having identical semantics return same output. Also, $\mathbf{Sem}_{1a}^{mat}$ has the disadvantage that deleting inexistent triples might still have side-effects. Combining $\mathbf{Sem}_{1a}^{mat}$ and $\mathbf{Sem}_2^{mat}$, namely $\mathbf{Sem}_3^{mat}$, deemed to be too aggressive in case of deletions, and thus fulfilling the least number of postulates.

The same holds for reduce-preserving semantics, i.e., one would always prefer $\mathbf{Sem}_0^{red}$, unless one's motivation is to delete the causes – in that case use $\mathbf{Sem}_1^{red}$.

For TBox updates, we see (cf. Table 42) that both semantics fulfill the exact number of postulates as $\mathbf{Sem}_2^{mat}$, this is not surprising given the similarity in their respective definitions. Also, discussing a semantics which combines both ABox and TBox is a next step that would make sense to investigate in the future, not only in the context of RDFS but also in the more expressive RDFS$_\neg$.

Next, continuing with "from theory to practice" approach, our preliminary implementation shows the feasibility of the proposed approaches on top of an off-the-shelf triple store, namely Apache TDB. As expected and confirmed by our experiments, those semantics that rely on rewriting ($\mathbf{Sem}_2^{mat}$), especially the optimised ones, in general perform better than the ones using materialisation. Regarding the comparison between mat-preserving vs. red-preserving, from the experiments we can conclude that keeping the store reduced could be a viable strategy, given that reduce operator can be computed using SPARQL 1.1 update query, and subsequently rewriting queries does not add dramatic costs.

Our work can be trivially extended to any standard compliant, off-the-shelf triple store such as RDF4J, and different benchmark datasets used in OBDM setting such as NPD [LRS+14] to name a few.

As for further related works, in the context of reduced stores, we refer to [PPSW13], where the cost of redundancy elimination under various (rule-based) entailment regimes,

including RDFS, is discussed in detail.

8.2.2 *DL-Lite*$_{\text{rdfs}\neg}$

As a stepping stone, we answered the research questions in Sec. 1.4, by proposing different semantics of SPARQL 1.1 Update in the context of interplay with *DL-Lite*$_{\text{RDFS}\neg}$:

> Given a SPARQL update posed over a triple store, how can we preserve consistency as well as materialisation of the triple store? Which triples (belonging to the update vs triple store) should be given priority when resolving inconsistencies?

The idea has been to extend further in the context of *DL-Lite*, building upon thoroughly studied query rewriting techniques (not necessarily relying on materialisation), and at the same time benefit from a more expressive ontology language. Expanding beyond our simple minimal RDFS language towards more features of *DL-Lite* or coverage of unrestricted RDF graphs would impose new challenges: for instance, consistency checking and consistency-preserving updates, which do not yet play a role in the setting of RDFS, would become relevant.

Thus, in this dissertation we combined SPARQL Update and RDFS entailment by adding disjointness axioms as a first step towards dealing with inconsistencies in the context of SPARQL Updates. We distinguish the case of intrinsic inconsistency, localized within instantiations of the INSERT clause of a SPARQL update, and the usual case when the new information is inconsistent with the old knowledge. In the former case, our solution was to discard all solutions of the WHERE query that participate in an inconsistency. For the latter case, we discussed several reconciliation strategies, well suited for efficient implementation in SPARQL.

Intrinsic consistency of an update is a common assumption in knowledge base update (e.g. [CKNZ10, LLMW06, DLPR09, FKAC13]), which can be easily violated in the case of SPARQL updates. As we have shown, this assumption is not trivially verifiable in the context of SPARQL Updates where DELETE/INSERT atoms are instantiated by a WHERE clause, and clashing triples could be instantiated within the same INSERT operation. It is worth noting that our resolution strategy for intrinsic inconsistency called safe rewriting can be combined with all three update semantics using just the basic SPARQL operators.

Next, taken that the problem of intrinsic consistency is solved, we have demonstrated how to extend the approach of [CKNZ10] to SPARQL updates. We have defined a materialisation and consistency preserving rewriting for SPARQL updates that essentially combines the ideas of [CKNZ10] and the previous work on SPARQL updates under RDFS for materialised triple stores, dealing with clashes due to class disjointness axioms in a brave manner. That is, we overwrite inconsistent information in the triple store in favor of information being inserted. Alternatively, we have also defined a dual consistency-reserving update semantics that on the contrary discards insertions that would lead to inconsistencies. Furthermore, we have also shown how a compromise between the two

extreme approaches can be defined and implemented using SPARQL's partial bindings feature.

Regarding the postulates, we investigated how brave, cautious and fainthearted semantics do in terms of fulfilling them. We can see from the results (cf. Table 52) that $\mathbf{Sem}^{mat}_{brave}$ seem to be the most rational, i.e., fulfill the most number of postulates. This is down to the fact that it can be viewed as the most natural that one would use in the case of handling negative information. $\mathbf{Sem}^{mat}_{caut}$ and $\mathbf{Sem}^{mat}_{faint}$ fulfill one postulate less than $\mathbf{Sem}^{mat}_{brave}$.

Same as in the case of *DL-Lite*$_{\text{RDFS}}$, our preliminary implementation shows the feasibility of all proposed approaches on top of an off-the-shelf triple store supporting SPARQL and SPARQL update (Apache TDB). As argued in Sec. 7.2, cautious semantics is always worse than brave semantics because of the additional ASK query, whereas between brave and fainthearted depends from the actual rewritings, i.e., based on the TBox axioms. In other words, if inserts produce a lot of clashes, then certainly brave semantics would perform worse than fainthearted; in the other case of low clashes then checks in WHERE condition of the fainthearted semantics would take more time.

As regards to further related work, the consideration of negative information is an important issue also in other related works on knowledge base updates: for instance, the seminal work on database view maintenance by Gupta et al. [GMS93] is also used in the context of materialised views using Datalog rules with stratified negation. Likewise, let us mention the work of Winslett[Win05] on formula-based semantics to updates, where negation is also considered.

As argued in Sec. 5.3.1, brave semantics implements the most established approach of adapting the new information fully via a minimal change, which is feasible in the setting of fixed RDFS$_\neg$ TBoxes. Also semantics deliberating between accepting and discarding change are known (see [Han99] for a survey). In [NRG12] an approach involving user interaction to decide whether to accept or reject an individual axiom is considered, with some part of the update being computed automatically in order to ensure its consistency. In this setting are not covered interactive procedures (although they clearly make sense in the case of more complex TBoxes or for TBox updates – like in (OWL) DBpedia). Instead, we rely on the resolution strategies which are simple for the user to understand and can be efficiently encoded in SPARQL. In a practical KB editing system, one should probably combine the two approaches, e.g. for resolving the intrinsic inconsistency. Likewise, the approaches [BBPW14], [FKAC13] and [KZC13] consider grounded updates only, whereas our focus is on implementation of updates in SPARQL. The approach in [FKAC13] captures RDFS and several additional types of constraints and is close in spirit to our brave semantics.

As future work, much interesting work remains to be done in order to optimize rewritten updates. Moreover, we plan to further extend our work towards increasing coverage of more expressive logics and OWL profiles, namely additional axioms from OWL 2 RL or OWL 2 QL [MGH$^+$12]. Also, as it is the case with *DL-Lite*$_{\text{RDFS}}$, the extension of

experiments with different triple stores and benchmark datasets is left for future work.

8.2.3 (OWL) DBpedia

As a final piece of work, we addressed the research questions in Sec. 1.5, by leveraging different semantics of SPARQL 1.1 Update we have defined for dealing with inconsistencies ($DL\text{-}Lite_{\text{RDFS}_\neg}$), and lifting them in the context of (OWL) DBpedia and mappings:

> Given a SPARQL update, how can we resolve mappings and translate it into updates of the underlying DBpedia infoboxes? Or, the other way around, can any of the update semantics discussed before capture an infobox update?

We presented first insights to allow ontology-based updates of wiki content. This was used as an use case for updates in the context of Ontology-Based Data Management, in the setting where mappings are present and TBox is more expressive. For this work, we have adopted the update semantics from the previous section that are able to handle inconsistencies.

The work was motivated by the little attention that has been paid to the benefits that the semantic infrastructure can bring to maintain the wiki content, for instance to ensure that the effects of a wiki edit are consistent across infoboxes.

The ultimate aim is not to compare our approach with any belief revision operators, but rather using/developing statistics (pre-existing data) and patterns (pre-existing interactions) as a means for helping users in making a meaningful choice, complementing work on belief revision with practical guidelines.

The main distinguishing characteristic herein is the DBpedia OBDM setting, and the focus on update accommodation strategies which are simple, comprehensible for the user and can draw from pre-existing meta-knowledge, such as already existing mapping patterns resp. usage frequencies of certain infobox fields, to decide update ambiguities upon similar, prototypical objects in the underlying data, estimating probabilities of alternative update translations. Our goal in this work was twofold: on the one hand, to understand and to formalize the DBpedia setting from the OBDM perspective, and on the other hand, to explore more pragmatic approaches to OBDM. To the best of our knowledge, it is the first attempt to study DBpedia mappings from the formal point of view. We found out that although the worst-case complexity of OBDM can be prohibitively high (even with low expressivity ontology and mapping languages), the real data, mappings and ontology found in DBpedia do not necessarily hit this full potential complexity; indeed, we conclude that the study and development of best-effort pragmatic approaches—some of which we have explored—is worthwhile.

We realized that various worst-case scenarios of update translation, especially those exhibiting the intractability of update handling, can be hardly realized in the current DBpedia version (mappings and ontology). From the practical point of view, the following aspects of OBDM appear crucial for the DBpedia case. Firstly, it is the inherent ambiguity of update translation; mappings often create a many-to-one or many-to-many

relationships between infobox and DBpedia properties. Second, concisely presenting a large number of options to the user is a challenge, hence an automatic selection of most likely update translations is likely required. Finally, being a curated system, Wiki also requires curated updates. Thus, splitting a SPARQL update into small independent pieces to be verified by the Wiki maintainers is needed as well. Note that human intervention is often unavoidable, since calculating mappings involve non-invertible functions.

Our early practical experiments with a DBpedia-based OBDM prototype shows that high worst case complexity of update translation can have little to do with actual challenges of OBDM for curated data. Rather, simple and comprehensible update resolution policies, reliable methods of confidence estimation and the ability to automatically learn and use best practices should be considered.

As further related work, the majority of existing OBDM approaches (e.g., [PCS14, KRRM$^+$14, RMR15]) consider the problem of query answering only rather than updates, using different fragments of OWL. The emphasis in those approaches is in algorithms for query rewriting considering one-to-many, many-to-many mappings, where queries consist of also variables (without an instantiation step as in our case).

As for updates and tgds, the approach [KGIT13] addresses a quite different setting of a peer data network in which data and updates are propagated via tgds. The peers in the network do not impose additional schema constraints (like the DBpedia TBox), features like class disjointness are not part of the setting, the focus is on combining the external data with local updates in a peer network.

We mentioned work reporting about inconsistencies in DBpedia [BKPR14, DKF$^+$15]. In another work about detecting inconsistencies within DBpedia [PG15] have considered mappings to the DOLCE upper ontology to detect even more inconsistencies, operating in a more complex ontology language using a full OWL DL reasoner (HermiT). Their approach is orthogonal in the sense that they focus on detecting and resolving systematic errors in the DBpedia ontology itself, rather than automatically fixing the assertions, leave alone the data in Wikipedia itself. Nonetheless, it would be an interesting future direction to combine these two orthogonal approaches.

It is also worth mentioning work in the domain of applying statistical methods for disambiguating updates, e.g., [TKS12], namely for enriching the TBox based on the data, which is actually not our scope, as we do not modify the TBox here.

As regards to updates where repairing mappings are considered to deal with potential inconsistencies, there is a preliminary work [LRS$^+$17] that considers updates both on source and ontology level.

Recently Wikipedia partially shifted to another, structured datasource than infoboxes, namely, Wikidata. We note that the model of Wikidata is different to DBpedia; different possible representations in plain RDF or Relational models have been recently suggested/discussed [HHR$^+$16]. Our approach could potentially help in bridging between the two, which we leave to future work. In that aspect, we have also designed and imple-

mented a method to compute similar entities in Wikidata, with the ultimate aim of capturing the relative completeness measure of entities.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ABK+07] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *Proc. of ISWC*, 2007.

[ACKZ14] Alessandro Artale, Diego Calvanese, Roman Kontchakov, and Michael Zakharyaschev. The dl-lite family and relations. *CoRR*, abs/1401.3487, 2014.

[AGM85] Carlos E. Alchourrón, Peter Gärdenfors, and David Makinson. On the logic of theory change: Partial meet contraction and revision functions. *J. Symb. Log.*, 50(2):510–530, 1985.

[AM82] Carlos E. Alchourron and David Makinson. On the logic of theory change: Contraction functions and their associated revision functions. *Theoria*, 48:14–37, 1982.

[BBL05] Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the el envelope. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, pages 364–369, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

[BBLPC13] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. Turtle – Terse RDF Triple Language. W3C Candidate Recommendation, February 2013. `http://www.w3.org/TR/2013/CR-turtle-20130219/`.

[BBPW14] Salem Benferhat, Zied Bouraoui, Odile Papini, and Eric Würbel. A prioritized assertional-based revision for *DL-Lite* knowledge bases. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence*, volume 8761 of *Lecture Notes in Computer Science*, pages 442–456. Springer, 2014.

[BCC+16] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk, and Guohui Xiao. OBDA beyond relational dbs: A study for mongodb. In *Proceedings of the 29th International Workshop on Description Logics, Cape Town, South Africa, April 22-25, 2016.*, 2016.

[BCM+03] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003.

[Ber03] Philip A. Bernstein. Applying model management to classical meta data problems, 2003.

[BGe04] Dan Brickley, R.V. Guha, and (eds.). RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, W3C, 2004.

[BKO+11] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. OWLIM: A family of scalable semantic repositories. *Semantic Web*, 2(1):33–42, 2011.

[BKPR14] Stefan Bischof, Markus Krötzsch, Axel Polleres, and Sebastian Rudolph. Schema-agnostic query rewriting in SPARQL 1.1. In *Proc. of the 13th Int'l Semantic Web Conf. (ISWC 2014)*. Springer, October 2014.

[BKVH02] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *The Semantic WebISWC 2002*, pages 54–68. Springer, 2002.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[BRN18] Vevake Balaraman, Simon Razniewski, and Werner Nutt. Recoin: Relative completeness in wikidata. In *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*, pages 1787–1792, 2018.

[BS81] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, December 1981.

[BtCLW13] Meghyn Bienvenu, Balder ten Cate, Carsten Lutz, and Frank Wolter. Ontology-based data access: a study through disjunctive datalog, csp, and MMSNP. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 213–224, 2013.

[CCD+13] Cristina Civili, Marco Console, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Lorenzo Lepore, Riccardo Mancini, Antonella Poggi, Riccardo Rosati, Marco Ruzzi, Valerio Santarelli, and Domenico Fabio Savo. MASTRO STUDIO: managing ontology-based data access applications. *PVLDB*, 6(12):1314–1317, 2013.

[CCK+17] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Ontop: Answering SPARQL queries over relational databases. *Semantic Web*, 8(3):471–487, 2017.

[CDGL+07] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *Journal of Automated Reasoning*, 39(3):385–429, 2007.

[CDGL+08] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Path-based identification constraints in description logics. In *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning*, KR'08, pages 231–241. AAAI Press, 2008.

[CFG+12] G. Cong, W. Fan, F. Geerts, J. Li, and J. Luo. On the complexity of view update analysis and its application to annotation propagation. *IEEE Transactions on Knowledge and Data Engineering*, 24(3):506–519, March 2012.

[CKNZ10] Diego Calvanese, Evgeny Kharlamov, Werner Nutt, and Dmitriy Zheleznyakov. Evolution of DL-Lite knowledge bases. In *ISWC*, pages 112–128, 2010.

[CTS11] Alexandros Chortaras, Despoina Trivela, and Giorgos B. Stamou. Optimized query rewriting for OWL 2 QL. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pages 192–206, 2011.

[CW94] Stefano Ceri and Jennifer Widom. Deriving incremental production rules for deductive data. *Inf. Syst.*, 19(6):467–490, 1994.

[DA16] Andrea Dessi and Maurizio Atzori. A machine-learning approach to ranking RDF properties. *Future Generation Comp. Syst.*, 54:366–377, 2016.

[DB82] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, September 1982.

[DKF+15] Anastasia Dimou, Dimitris Kontokostas, Markus Freudenberg, Ruben Verborgh, Jens Lehmann, Erik Mannens, Sebastian Hellmann, and Rik Van de Walle. Assessing and refining mappings to RDF to improve dataset quality. In *The Semantic Web - ISWC 2015 - 14th Int'l Semantic Web Conf., Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*, volume 9367, pages 133–149. Springer, 2015.

[DLO+17] Giuseppe De Giacomo, Domenico Lembo, Xavier Oriol, Domenico Fabio Savo, and Ernest Teniente. Practical update management in ontology-based data access. In *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, pages 225–242, 2017.

[DLPR09] Giuseppe De Giacomo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. On instance-level update and erasure in description logic ontologies. *J. Log. Comput.*, 19(5):745–770, 2009.

[DORS16] Giuseppe De Giacomo, Xavier Oriol, Riccardo Rosati, and Domenico Fabio Savo. Updating dl-lite ontologies through first-order queries. In *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part I*, pages 167–183, 2016.

[EK12] Vadim Eisenberg and Yaron Kanza. D2rq/update: Updating relational data via virtual rdf. *WWW'12 - Proceedings of the 21st Annual Conference on World Wide Web Companion*, 04 2012.

[FG13] Enrico Franconi and Paolo Guagliardo. Effectively updatable conjunctive views. In *Proceedings of the 7th Alberto Mendelzon International Workshop on Foundations of Data Management, Puebla/Cholula, Mexico, May 21-23, 2013.*, 2013.

[FGM+13] Enrico Franconi, Claudio Gutierrez, Alessandro Mosca, Giuseppe Pirrò, and Riccardo Rosati. The logic of extensional RDFS. In *International Semantic Web Conference (ISWC2013)*, volume 8218 of *LNCS*, pages 101–116. Springer, October 2013.

[FHH+06] Ariel Fuxman, Mauricio A. Hernández, C. T. Howard Ho, Renée J. Miller, Paolo Papotti, and Lucian Popa. Nested mappings: Schema mapping reloaded. In *VLDB*, pages 67–78, 2006.

[FKAC13] Giorgos Flouris, George Konstantinidis, Grigoris Antoniou, and Vassilis Christophides. Formal foundations for rdf/s kb evolution. *Knowledge and Information Systems*, 35(1):153–191, 2013.

[FMK+08] Giorgos Flouris, Dimitris Manakanatas, Haridimos Kondylakis, Dimitris Plexousakis, and Grigoris Antoniou. Ontology change: Classification and survey. *Knowl. Eng. Rev.*, 23(2):117–152, June 2008.

[GHKP12] Birte Glimm, Adian Hogan, Markus Krötzsch, and Axel Polleres. OWL: Yet to arrive on the web of data? In *WWW2012 Workshop on Linked Data on the Web*, 2012.

[GHV11] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. Updating RDFS: from theory to practice. In *8th Extended Semantic Web Conf. (ESWC 2011)*, 2011.

[GHVD03] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: Combining logic programs with description logic. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 48–57, New York, NY, USA, 2003. ACM.

[GJS76] M.R. Garey, D.S. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1(3):237 – 267, 1976.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD Int'l Conf. on Management of Data*, pages 157–166. ACM, 1993.

[GOH+13] Birte Glimm, Chimezie Ogbuji, Sandro Hawke, Ivan Herman, Bijan Parsia, Axel Polleres, and Andy Seaborne. SPARQL 1.1 Entailment Regimes, March 2013. W3C Recommendation.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[GPP13] Paul Gearon, Alexandre Passant, and Axel Polleres. SPARQL 1.1 Update, March 2013. W3C Recommendation.

[GRAS17] Luis Galárraga, Simon Razniewski, Antoine Amarilli, and Fabian M Suchanek. Predicting completeness in knowledge bases. *WSDM*, 2017.

[Han99] Sven Ove Hansson. A survey of non-prioritized belief revision. *Erkenntnis*, 50(2-3):413–427, 1999.

[Hay04] Patrick Hayes. RDF semantics, 2004. W3C Recommendation.

[HD92] John V. Harrison and Suzanne W. Dietrich. Maintenance of materialized views in a deductive database: An update propagation approach. In *Proceedings of the Workshop on Deductive Databases held in conjunction with the Joint International Conference and Symposium on Logic Programming, Washington, D.C., USA, Saturday, November 14, 1992*, pages 56–65, 1992.

[HG11] Antonio Garrote Hernández and María N. Moreno García. Restful writable apis for the web of linked data using relational storage solutions. In Christian Bizer, Tom Heath, Tim Berners-Lee, and Michael Hausenblas, editors, *WWW2011 Workshop on Linked Data on the Web, Hyderabad, India, March 29, 2011*, volume 813 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[HHR+16] Daniel Hernández, Aidan Hogan, Cristian Riveros, Carlos Rojas, and Enzo Zerega. Querying wikidata: Comparing sparql, relational and graph

databases. In *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part II*, pages 88–103, 2016.

[HPS14] Patrick Hayes and Peter Patel-Schneider. RDF 1.1 semantics. W3C Recommendation, W3C, February 2014.

[HRG10] Matthias Hert, Gerald Reif, and Harald C. Gall. Updating relational data via sparql/update. In *Proceedings of the 2010 EDBT/ICDT Workshops*, EDBT '10, pages 24:1–24:8, New York, NY, USA, 2010. ACM.

[HS13] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language, March 2013. W3C Recommendation.

[JSIB10] Anja Jentzsch, Christopher Sahnwaldt, Robert Isele, and Christian Bizer. Dbpedia mapping language. Technical report, 2010.

[KBB11] Jakub Kotowski, François Bry, and Simon Brodt. Reasoning as axioms change - incremental view maintenance reconsidered. In *5th International Conference on Web Reasoning and Rule Systems (RR 2011)*, volume 6902 of *LNCS*, pages 139–154, Galway, Ireland, August 2011. Springer.

[Kel85] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '85, pages 154–163, New York, NY, USA, 1985. ACM.

[KGIT13] Grigoris Karvounarakis, Todd J. Green, Zachary G. Ives, and Val Tannen. Collaborative data sharing via update exchange and provenance. *ACM Trans. Database Syst.*, 38(3):19:1–19:42, September 2013.

[KM89] Hirofumi Katsuno and Alberto O. Mendelzon. A unified view of propositional knowledge base updates. In *IJCAI*, pages 1413–1419, 1989.

[KPSS14] Phokion G. Kolaitis, Reinhard Pichler, Emanuel Sallinger, and Vadim Savenkov. Nested dependencies: structure and reasoning. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 176–187, 2014.

[KRMZ13] Roman Kontchakov, Mariano Rodriguez-Muro, and Michael Zakharyaschev. Ontology-based data access with databases: A short course. In *Reasoning Web 2013*, volume 8067 of *LNCS*, pages 194–229. Springer, 2013.

[KRRM+14] Roman Kontchakov, Martin Rezk, Mariano Rodríguez-Muro, Guohui Xiao, and Michael Zakharyaschev. Answering sparql queries over databases under owl 2 ql entailment regime. In *Proc. of the 13th Int.l Semantic Web Conference - Part I*, ISWC '14, pages 552–567, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[Kr12] Markus Krötzsch. OWL 2 Profiles: An introduction to lightweight ontology languages. In Thomas Eiter and Thomas Krennwallner, editors, *Proceedings of the 8th Reasoning Web Summer School, Vienna, Austria, September 3–8 2012*, volume 7487 of *LNCS*, pages 112–183. Springer, 2012.

[KZC13] Evgeny Kharlamov, Dmitriy Zheleznyakov, and Diego Calvanese. Capturing model-based ontology evolution at the instance level: The case of *DL-Lite. J. Comput. Syst. Sci.*, 79(6):835–872, September 2013.

[Len02] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 233–246, New York, NY, USA, 2002. ACM.

[Len18] Maurizio Lenzerini. Managing data through the lens of an ontology. *AI Magazine*, 39(2):65–74, 2018.

[Lib14] Leonid Libkin. Certain answers as objects and knowledge. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*, 2014.

[LIJ+14] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Chris Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*, 2014.

[LIJ+15] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, et al. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[LLMW06] Hongkai Liu, Carsten Lutz, Maja Milicic, and Frank Wolter. Updating description logic aboxes. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *KR*, pages 46–56. AAAI Press, 2006.

[LRS+14] Davide Lanti, Martin Rezk, Mindaugas Slusnys, Guohui Xiao, and Diego Calvanese. The NPD benchmark for OBDA systems. In *Proceedings of the 10th International Workshop on Scalable Semantic Web Knowledge*

Base Systems co-located with 13th International Semantic Web Conference (ISWC 2014), Riva del Garda, Italy, October 20, 2014., pages 3–18, 2014.

[LRS+17] Domenico Lembo, Riccardo Rosati, Valerio Santarelli, Domenico Fabio Savo, and Evgenij Thorstensen. Mapping repair in ontology-based data access evolving systems. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 1160–1166, 2017.

[LS11] Maurizio Lenzerini and Domenico Fabio Savo. On the evolution of the instance level of dl-lite knowledge bases. In *Proceedings of the 24th International Workshop on Description Logics (DL 2011), Barcelona, Spain, July 13-16, 2011*, 2011.

[LSTW13] Carsten Lutz, Inanç Seylan, David Toman, and Frank Wolter. The combined approach to OBDA: taming role hierarchies using filters. In *The Semantic Web - ISWC 2013 - 12th Int'l Semantic Web Conf., Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part I*, pages 314–330, 2013.

[MAHP11] Alejandro Mallea, Marcelo Arenas, Aidan Hogan, and Axel Polleres. On Blank Nodes. In *Proceedings of the 10th International Semantic Web Conference (ISWC 2011)*, volume 7031 of *LNCS*. Springer, October 2011.

[MGH+12] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. Owl 2 web ontology language profiles (second edition). W3C Recommendation, W3C, December 2012. Available at `http://www.w3.org/TR/owl2-profiles/`.

[Mot07] Boris Motik. On the Properties of Metamodeling in OWL. *Journal of Logic and Computation*, 17(4):617–637, 2007.

[MPG07] Sergio Muñoz, Jorge Pérez, and Claudio Gutiérrez. Minimal deductive systems for RDF. In *ESWC*, pages 53–67, 2007.

[MRN16] Paramita Mirza, Simon Razniewski, and Werner Nutt. Expanding Wikidata's parenthood information by 178%, or how to mine relation cardinalities. *ISWC Posters & Demos*, 2016.

[NK04] Natalya Fridman Noy and Michel C. A. Klein. Ontology evolution: Not the same as schema evolution. *Knowl. Inf. Syst.*, 6(4):428–440, 2004.

[NPM+15] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. Rdfox: A highly-scalable RDF store. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*, pages 3–20, 2015.

[NRG12] Nadeschda Nikitina, Sebastian Rudolph, and Birte Glimm. Interactive ontology revision. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1213:118 – 130, 2012. Reasoning with context in the Semantic Web.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):Article 16 (45 pages), 2009.

[Pau17] Heiko Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic Web*, 8(3):489–508, 2017.

[PCS14] Freddy Priyatna, Oscar Corcho, and Juan Sequeda. Formalisation and experiences of r2rml-based sparql to sql query translation using morph. In *Proc. of the 23rd Int'l Conf. on World Wide Web*, WWW '14, pages 479–490. ACM, 2014.

[PDRN16] Radityo Eko Prasojo, Fariz Darari, Simon Razniewski, and Werner Nutt. Managing and consuming completeness information for wikidata using COOL-WD. *COLD*, 2016.

[Pep08] Pavlos Peppas. Belief revision. In *Handbook of Knowledge Representation*, pages 317–359. 2008.

[PFH06] Axel Polleres, Cristina Feier, and Andreas Harth. *Rules with Contextually Scoped Negation*. 2006.

[PG15] Heiko Paulheim and Aldo Gangemi. Serving dbpedia with dolce — more than just adding a cherry on top. In *Proc. of the 14th Int'l Conf. on The Semantic Web - ISWC 2015*, pages 180–196, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[PHDU13] Axel Polleres, Aidan Hogan, Renaud Delbru, and Jürgen Umbrich. RDFS & OWL reasoning for linked data. In *Reasoning Web 2013*, volume 8067 of *LNCS*, pages 91–149. Springer, July 2013.

[PLC+08] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. In Stefano Spaccapietra, editor, *Journal on Data Semantics X*, volume 4900 of *LNS*, pages 133–173. Springer, 2008.

[PMH09] Héctor Pérez-Urbina, Boris Motik, and Ian Horrocks. A comparison of query rewriting techniques for dl-lite. In *Proceedings of the 22nd International Workshop on Description Logics (DL 2009), Oxford, UK, July 27-30, 2009*, 2009.

[Pol07] Axel Polleres. From sparql to rules (and back). In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 787–796, New York, NY, USA, 2007. ACM.

[PPSW13] Reinhard Pichler, Axel Polleres, Sebastian Skritek, and Stefan Woltran. Complexity of redundancy detection on RDF graphs in the presence of rules, constraints, and queries. *Semantic Web – Interoperability, Usability, Applicability*, 4(4), 2013.

[RA10] Riccardo Rosati and Alessandro Almatelli. Improving query answering over dl-lite ontologies. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*, 2010.

[RE03] M Andrea Rodríguez and Max J. Egenhofer. Determining semantic similarity among entity classes from different ontologies. *TKDE*, 15(2):442–456, 2003.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[RKK+10] Sunitha Ramanujam, Vaibhav Khadilkar, Latifur Khan, Steven Seida, Murat Kantarcioglu, and Bhavani M. Thuraisingham. Bi-directional translation of relational data into virtual rdf stores. In *ICSC*, pages 268–276, 2010.

[RMR15] Mariano Rodríguez-Muro and Martin Rezk. Efficient sparql-to-sql with r2rml mappings. *Web Semantics: Science, Services and Agents on the World Wide Web*, 33(1), 2015.

[RN10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[RSN16] Simon Razniewski, Fabian M Suchanek, and Werner Nutt. But what do we actually know. *AKBC*, 2016.

[Seq16] Juan Federico Sequeda. *Integrating relational databases with the semantic web*. PhD thesis, University of Texas at Austin, TX, USA, 2016.

[SKW07] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A core of semantic knowledge. In *Proc. of the 16th Int'l Conf. on World Wide Web*, WWW '07, pages 697–706. ACM, 2007.

[SL12] Martin Slota and João Leite. A unifying perspective on knowledge updates. In *Logics in Artificial Intelligence*, volume 7519 of *Lecture Notes in Computer Science*, pages 372–384. Springer Berlin Heidelberg, 2012.

[TKS12] Gerald Töpper, Magnus Knuth, and Harald Sack. Dbpedia ontology enrichment for inconsistency detection. In *Proc. of I-SEMANTICS '12*, pages 33–40. ACM, 2012.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[UMJ+13] Jacopo Urbani, Alessandro Margara, Ceriel J. H. Jacobs, Frank van Harmelen, and Henri E. Bal. Dynamite: Parallel materialization of dynamic rdf data. In *International Semantic Web Conference (ISWC2013)*, volume 8218 of *LNCS*, pages 657–672. Springer, October 2013.

[VK14] Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

[VSM05] Raphael Volz, Steffen Staab, and Boris Motik. Incrementally maintaining materializations of ontologies stored in logic databases. *J. on Data Semantics*, 2:1–34, 2005.

[Win05] Marianne Winslett. *Updating Logical Databases.* Cambridge University Press, 2005.

[XCK+18] Guohui Xiao, Diego Calvanese, Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati, and Michael Zakharyaschev. Ontology-based data access: A survey. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 5511–5519. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[ZKNC19] Dmitriy Zheleznyakov, Evgeny Kharlamov, Werner Nutt, and Diego Calvanese. On expansion and contraction of dl-lite knowledge bases. *J. Web Semant.*, 57, 2019.