Imperial College London
Department of Computing

Inference as a Data Management Problem

Yu Liu

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College and
the Diploma of Imperial College, October 2016

Inference as a Data Management Problem

Yu Liu

Supervised by Peter McBrien and Peter Pietzuch

October 2016

Declaration of Originality

I declare that this thesis is my own original work, and I have fully cited and referenced all material and results that are not original to this work.

Abstract

Inference over OWL ontologies with large A-Boxes has been researched as a data management problem in recent years. This work adopts the strategy of applying a tableaux-based reasoner for complete T-Box classification, and using a rule-based mechanism for scalable A-Box reasoning. Specifically, we establish for the classified T-Box an inference framework, which can be used to compute and materialise inference results. The inference we focus on is type inference in A-Box reasoning, which we define as the process of deriving for each A-Box instance its memberships of OWL classes and properties. As our approach materialises the inference results, it in general provides faster query processing than non-materialising techniques, at the expense of larger space requirement and slower update speed. When the A-Box size is suitable for an RDBMS, we compile the inference framework to triggers, which incrementally update the inference materialisation from both data inserts and data deletes, without needing to recompute the whole inference. More importantly, triggers make inference available as atomic consequences of inserts or deletes, which preserves the ACID properties of transactions, and such inference is known as transactional reasoning. When the A-Box size is beyond the capability of an RDBMS, we then compile the inference framework to Spark programmes, which provide scalable inference materialisation in a Big Data system, and our evaluation considers up to reasoning 270 million A-Box facts. Evaluating our work, and comparing with two state-of-the-art reasoners, we empirically verify that our approach is able to perform scalable inference materialisation, and to provide faster query processing with comparable completeness of reasoning.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor Dr. Peter McBrien for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

Secondly, I would like to thank my second supervisor Dr. Peter Pietzuch for his feedback on my early stage report and the help he has offered afterwards.

Thirdly, I would like to thank my colleague Ms. Lama Al Khuzayem for her encouragement and suggestions. I would also like to thank all the 558c officemates, Dr. Daniel Sykes, Ms. Juliana Patricia Vicente Franco, Mr. Fergus Leahy, Dr. Yasmin Rafiq, Dr. Chris Novakovic and Mr. Sylvan Clebsch for enjoying the warmest and happiest office together with me.

Last but not the least, I would like to thank my parents for supporting me spiritually throughout writing this thesis and my life in general.

Contents

Abstract **9**

Acknowledgements **11**

1 Introduction **23**

1.1 Motivation and Objectives 23

1.2 Contributions 30

1.3 Thesis Outline 31

2 Review of Web Ontology Language **35**

2.1 Introduction 35

2.2 OWL 2 Overview 36

2.2.1 Terminological Box 36

2.2.2 Assertional Box 37

2.2.3 Inference 38

2.3 The Direct Semantics of OWL 2 40

2.3.1 Interpretation $\mathcal{I}$ of OWL 2 DL 41

2.3.2 Class-related Semantics 42

2.3.3 Property-related Semantics 50

2.4 Summary 57

3 Review of Inference **59**

3.1 Introduction 59

3.2 Inference Tasks 60

3.2.1 Global Inference Tasks 60

3.2.2 Local Inference Tasks 62

3.2.3 Inference Properties of Reasoners 63

3.3 Tableaux-based Inference 64

3.3.1 $\mathcal{ALC}$ ontologies 64

3.3.2 Tableaux algorithm 65

3.4 Rule-based Inference 69

3.4.1 OWL 2 Profiles 70

3.4.2 Query-rewriting Approach 74

3.4.3 Materialised Approach 76

3.5 Summary 77

4 Type Inference from Data Inserts **79**

4.1 Introduction 79

4.2 Motivating Example 81

4.3 SQOWL2 Overview 83

4.3.1 SQOWL2 Architecture 84
4.3.2 Type Inference from Data Inserts by Triggers 85
4.3.3 SQOWL2 Features 87
4.4 Establishing the ATIDB Schema 88
4.4.1 Canonical Schema 88
4.5 Generating Logical Triggers 89
4.5.1 OWL Classes and Properties 90
4.5.2 The Semantics of Class Axioms 90
4.5.3 The Semantics of Property Axioms 99
4.6 Optimisation 107
4.6.1 Canonical Schema to Conventional Schema 108
4.7 Summary 109

5 Type Inference from Data Deletes 111

5.1 Introduction 111
5.2 Motivating Example 112
5.3 Review of Incremental View Maintenance 114
5.3.1 Counting Algorithm 114
5.3.2 DRed Algorithm 116
5.4 SQOWL2 for Data Deletes 117
5.4.1 The State of Each Data Item 118
5.4.2 Ontology Inserts/Deletes & Reasoner Inserts/Deletes 119

5.4.3 Materialising Type Inference with Data State . . . 121

5.4.4 Label & Check for Data Deletes . . . 123

5.5 Triggers and Inference Rules for OWL 2 RL axioms . . . 126

5.5.1 OWL Classes and Properties . . . 126

5.5.2 The Semantics of Class Axioms . . . 128

5.5.3 The Semantics of Property Axioms . . . 133

5.6 Soundness & Completeness of SQOWL2 . . . 137

5.7 Summary . . . 138

6 Type Inference over Big Data 139

6.1 Introduction . . . 139

6.2 Review of Spark . . . 142

6.2.1 RDD & DAG scheduler . . . 142

6.2.2 Spark vs. Hadoop . . . 144

6.3 SPOWL Overview . . . 145

6.3.1 SPOWL Architecture . . . 145

6.3.2 Approach Demonstration . . . 146

6.4 Materialising Inference Closure for OWL 2 RL . . . 147

6.4.1 Compiling for OWL 2 RL Class-related Axioms . . . 148

6.4.2 Compiling for OWL 2 RL Property-related Axioms . . . 155

6.4.3 Optimised Order of Executing Spark Programmes . . . 159

6.5 Summary . . . 162

7 Implementation & Evaluation 165

7.1 Introduction . . . 165

7.2 Implementation of SQOWL2 . . . 166

7.2.1 Two Schemas to Represent Data and State . . . 166

7.2.2 Implementing Logical Triggers as Physical Triggers . . . 169

7.2.3 Optimisation in SQOWL2 . . . 176

7.3 Evaluation of SQOWL2 . . . 177

7.3.1 Performance of Incremental Type Inference . . . 179

7.3.2 Performance of Query Processing . . . 181

7.4 Implementation of SPOWL . . . 187

7.4.1 Spark Application Initialisation . . . 188

7.4.2 Initial Data Loading . . . 188

7.4.3 Physical Spark Programmes Generation . . . 190

7.4.4 Optimisation in SPOWL . . . 191

7.5 Evaluation of SPOWL . . . 192

7.5.1 Performance of Inference Materialisation . . . 193

7.5.2 Performance of Query Processing . . . 194

7.6 Summary . . . 196

8 Conclusion 199

8.1 Summary of Thesis Achievements . . . 201

8.2 Future Work . . . 202

References **203**

A List of Abbreviations **217**

B Theorem PR1 **219**

List of Tables

3.1 The family of DL acronyms 65

3.2 $\mathcal{ALC}$ to NNF 67

3.3 Class & property expressions and restrictions in OWL 2 QL and OWL 2 RL . . 72

3.4 Axioms supported in OWL 2 QL and OWL 2 RL 73

5.1 Triggers and inference rules generated from class axioms 129

5.2 Triggers and inference rules generated from property axioms 134

7.1 Data loading time (s) 179

7.2 Data loading speed (inserts/s) 179

7.3 Data deleting speed (deletes/s) 180

7.4 Average query processing speed (q/m) 183

7.5 Detailed query processing time (ms) 184

7.6 Query improvements (ms) by SQOWL2 after database tuning (results in Table 7.5 → improved performance) 186

7.7 Performance of Inference Materialisation by SPOWL 193

7.8 Performance of Query Processing by SPOWL 195

List of Figures

3.1 Handing ⊓, ⊔, ∀ and ∃ by tableaux algorithm 69

4.1 Categories of Type Inference Systems 80

4.2 $T_{4.1}$: inserting CzechBeer(CzechPaleLager) 82

4.3 $T_{4.2}$: inserting CzechLager(CzechDarkLager) 82

4.4 Approach Architecture for Type Inference in an RDBMS 84

5.1 Motivating Example of Type Inference from Data Deletes 113

5.2 Data state transitions 118

5.3 $T_{5.1}$: insert Ale(IrishRedAle) 121

5.4 $T_{5.2}$: insert IrishBeer(IrishRedAle) 122

5.5 $T_{5.3}$: delete Ale(IrishRedAle) 125

6.1 Job Scheduling between Hadoop (left) and Spark (right) 144

6.2 Approach Architecture for Type Inference in a Big Data System 145

6.3 Dependence among transformation rules 160

6.4 Acyclic property hierarchy 161

6.5 Cyclic property hierarchy 161

7.1 Front-end and back-end schemas 168

7.2 Before insert trigger on S_B.Ale 170

7.3 After insert trigger on S_B.Ale 171

7.4 After insert trigger on S_B.Beer 171

7.5 After insert trigger on S_B.LiquidBread 172

7.6 Before delete trigger on S_F.Ale 173

7.7 After update trigger on S_B.Ale 174

7.8 Before delete trigger on S_B.Ale 175

7.9 after delete trigger on S_B.Ale 175

7.10 Before delete trigger on S_B.Beer 176

7.11 LUBM Query 1 in SPARQL 181

7.12 LUBM Query 1 in SQL 182

7.13 The Components of a Spark application 188

7.14 Inference Materialisation by SPOWL 194

7.15 LUBM Query 1 in Spark 194

7.16 Average Query Processing by SPOWL 195

Chapter 1

Introduction

1.1 Motivation and Objectives

Modelling information and knowledge has been researched as the problem of **Knowledge Representation** (**KR**) [Sow00] since the 1970's. Many approaches, such as semantic networks [Leh92], frames [Sow14] and **Description Logic** (**DL**) [BCM+10], have emerged to provide a formalised and machine-manageable model for storing knowledge concepts and their relations as a knowledge base. Furthermore, applications are able to conduct a process of **inference**, which derives implicit consequences from the explicitly represented information and knowledge. Inference is often considered as the most significant feature of knowledge modelling compared to traditional data modelling, such as the **Entity-Relationship** (**ER**) model [BCN92], the **relational** model [Cod70] and the **Unified Modelling Language** (**UML**) modelling [Boo05].

The **Web Ontology Language** (**OWL**) [MVH04] endorsed by the **World Wide Web Consortium** (**W3C**) has become one of most popular KR languages during the last decade. OWL allows information and knowledge to be expressed as an **ontology**, and many knowledge fields adopt it for expressing their domain information. Well-known ontologies include DBpedia [ABK+07], Gene ontology [ABB+00], Music ontology [RASG07], *etc*. In an OWL ontology, atomic **individuals** can be classified into OWL **classes**, and be connected to each other by OWL **properties**. Moreover, OWL provides numerous **constructors** for ontology

designers to express various relations and constraints as **axioms**, such as subsumption, universal and existential quantifications, and key constraints. In particular, an ontology divides its represented knowledge and information into schema and data, which are contained in a **Terminological Box** (**T-Box**) and an **Assertional Box** (**A-Box**), respectively.

To illustrate OWL, we introduce a beer ontology[1] storing information and knowledge about beers. In its T-Box, we may have three OWL classes, CzechLager, Lager and CzechBeer respectively storing beer individuals which are Czech lagers, general lagers and Czech beers. We may define a property called hasFlavour to relate beer individuals to some individuals from another class Flavour. Besides OWL classes and properties, we can specify a more complex relation:

$$\mathsf{CzechLager} \equiv \mathsf{Lager} \sqcap \mathsf{CzechBeer} \tag{1.1}$$

which defines every individual of CzechLager is some beer which is both a Lager and a CzechBeer. This relation uses two OWL constructors EquivalentClasses and IntersectionOf (symbolised as $\equiv$ and $\sqcap$ in DL, respectively), and because it specifies some schema information of the beer ontology, it should be a T-Box axiom.

In addition to a T-Box (containing classes, properties and axioms), we can assert some known facts in an A-Box; for example, an individual CzechPaleLager can be specified as a type of Lager and a kind of CzechBeer by (1.2) and (1.3), respectively, and another individual CzechDarkLager can be defined as a member of CzechLager by (1.4).

$$\mathsf{Lager}(\mathsf{CzechPaleLager}) \tag{1.2}$$

$$\mathsf{CzechBeer}(\mathsf{CzechPaleLager}) \tag{1.3}$$

$$\mathsf{CzechLager}(\mathsf{CzechDarkLager}) \tag{1.4}$$

The most recent release OWL 2 [GHM+08] comes with a **Direct Semantics** [MPSG12] for interpreting its components, and inference can be computed based on the logical implications defined by the semantics. Inference tasks, such as computing all subsumption relations contained in a T-Box, are decidable if an ontology is interpreted by the Direct Semantics, because the underpinning $\mathcal{SROIQ}$ DL [Rud11] of the semantics is a decidable fragment of the **First-Order Logic** (**FOL**) [Bar77]. To automatically perform inference, the well-known **tableaux algorithm** [Rud11] has been widely adopted by **reasoners** (i.e. software applications which

[1] `https://github.com/yl12510/thesis/tree/master/beer.owl`

offer an inference service), such as Pellet [SPG+07] and Hermit [GHM+14]. The tableaux algorithm reduces every inference task to a problem of satisfiability checking, and verifies the satisfiability by attempting to establish a tableau. The tableaux algorithm is able to derive all correct logical consequences for the whole $\mathcal{SROIQ}$ DL, and thus tableaux-based reasoners are known to provide a sound and complete inference service.

Take the T-Box axiom (1.1) as an example, by using the tableaux algorithm, Pellet or Hermit infers three subsumption relations below from (1.1):

$$\mathsf{CzechLager} \sqsubseteq \mathsf{Lager} \tag{1.5}$$

$$\mathsf{CzechLager} \sqsubseteq \mathsf{CzechBeer} \tag{1.6}$$

$$\mathsf{Lager} \sqcap \mathsf{CzechBeer} \sqsubseteq \mathsf{CzechLager} \tag{1.7}$$

where (1.5) and (1.6) denote every individual of CzechLager is both a Lager and a CzechBeer, and (1.7) specifies some individual which belongs to both Lager and CzechBeer should be a CzechLager. The task of inferring all subsumption relations contained in a T-Box is called **classification**, and based on a classified T-Box, a tableaux-based reasoner is also able to determine whether an individual belongs to a particular class, which is known as another inference task **instance checking**. For example, since (1.4) asserts CzechDarkLager as a CzechLager, determining whether CzechDarkLager belongs to Lager should result in true because of (1.5).

However, expressing real-world knowledge domains may not necessarily require the full OWL 2 language when they have a simple knowledge schema (i.e. T-Box). Inference over the full OWL 2 is not tractable (i.e. a complexity beyond polynomial time); indeed, most inference tasks, such as *ontology consistency* and *class expression subsumption* have a complexity of N2EXPTIME[2]-complete [Krö12a]. In addition, the tableaux algorithm always suffers an issue of inefficiency when ontologies contain very large A-Boxes, since it needs to verify every A-Box fact for a particular inference task. This is unacceptable when considering most real-world ontologies, such as DBpedia and WordNet [Mil95], which usually contain huge numbers of A-Box facts (e.g. DBpedia has approximately 3 billion A-Box facts as of Sept. 2014, compared to a T-Box of around 1,200 classes, 2,900 properties and 7,000 logical axioms).

Therefore, research on providing tractable inference for ontologies having a simple T-Box and

[2]N2EXPTIME is the set of decision problems solvable by a non-deterministic algorithm in $\mathcal{O}(2^{2^n})$ time.

large A-Boxes has attracted much attention over the recent years, and the essential idea is to sacrifice the OWL expressivity for desirable computation properties. Indeed, OWL 2 provides three profiles [MGH+12], namely, OWL 2 EL, OWL 2 QL and OWL 2 RL, each of which is a subset of the full OWL 2, aiming for particular application scenarios. Among them, QL and RL especially focus on ontologies with large A-Boxes, while EL addresses large T-Boxes. The QL profile targets applications whose major inference task is query processing over large A-Boxes, and in theory it guarantees that conjunctive queries over OWL 2 QL ontologies can be answered in LOGSPACE[3] time by a suitable inference technique. On the other hand, the RL profile is specified for applications aiming at scalable inference over ontologies with large A-Boxes. From the viewpoint of inference techniques, **rule-based inference** [SS11], which uses some pre-defined reasoning rules for the derivation of implicit consequences, has been widely applied in many large-scale reasoners, such as DLDB [PZH08], Ontop [BCH+14], Oracle's RDF Store [WED+08] and OWLim [KOM05].

Rule-based inference can be further classified into **query-rewriting** [PUHM09] and **materialised** [MNP+14] approaches. Reasoners taking a query-rewriting approach (e.g. Stardog [PURDG+12], Ontop, DLDB) rewrite a query over an ontology into some sub-queries over the base facts in the ontology (according to some reasoning rules), such that not only explicit but also implicit answers to the query can be computed dynamically. Take the T-Box axiom (1.1) as an example again, a query-rewriting approach might use the reasoning rules below (in Datalog) to rewrite queries asking for individuals in CzechLager:

$$\mathsf{CzechLager}(x) \text{ :- } \mathsf{CzechLager}_e(x)$$

$$\mathsf{CzechLager}(x) \text{ :- } \mathsf{CzechLager}_i(x)$$

$$\mathsf{CzechLager}_i(x) \text{ :- } \mathsf{Lager}(x), \mathsf{CzechBeer}(x)$$

These reasoning rules specify that individuals of CzechLager should contain extensional and intensional parts (i.e. $\mathsf{CzechLager}_e(x)$ and $\mathsf{CzechLager}_i(x)$), where the extensional part should be A-Box facts explicitly asserted to CzechLager, and the intensional part should include individuals commonly existing in Lager and CzechBeer. Query-rewriting usually considers OWL 2 QL ontologies, and does not need to store implicit derivations because of the dynamic inference. However, this approach often becomes inefficient when a query is rewritten to many complex

[3]LOGSPACE is the set of decision problems solvable by a deterministic algorithm in $\mathcal{O}(\log(n))$ time.

sub-queries, and executed frequently.

When the number of the updates to an ontology is significantly fewer than the number of queries, it might be more efficient to adopt a materialised approach (e.g. Oracle's RDF Store, OWLim, WebPIE [UKM+12], RDFox [NPM+15], Minerva [ZML+06, LMZ+07]), which materialises inference results before asking a query; thus queries can be answered directly from the materialisation. For example, a materialised approach might translate the T-Box axiom (1.1) into the following **if**...**then**... rules:

$$\textbf{if } \mathsf{CzechLager}(x) \textbf{ then } \mathsf{Lager}(x) \text{ and } \mathsf{CzechBeer}(x)$$

$$\textbf{if } \mathsf{Lager}(x) \text{ and } \mathsf{CzechBeer}(x) \textbf{ then } \mathsf{CzechLager}(x)$$

which are used for computing the inference materialisation. Reasoners adopting a materialised approach usually consider OWL 2 RL ontologies and they require no computation of inference when a query is executed over their inference materialisation. However, materialising requires more space than non-materialising (e.g. query-rewriting), and how to materialise the inference results in a scalable manner, and how to efficiently update the materialisation when there are some updates to ontologies are two major challenges to a materialised approach.

The hypothesis of this PhD study is that a feasible approach for managing inference over large ontologies held in databases can be developed, such that inference can be performed in a scalable manner, and the cost of inference does not introduce an impractical overhead. We wish to develop an approach that can be used in cooperation with existing database applications, such as banking, search engines, digital advertising *etc.*, to enhance them with inference capabilities at an execution-time cost that is realistic to bear in such applications. Our hypothesis is that in such applications it is more sensible to adopt a materialised approach to inference, based on the assumption that queries in such applications are much more frequent than updates. Additionally, since most database applications use a small and static schema to model their large and dynamic data, we further assume that the T-Box of ontologies is much smaller compared to the size of the A-Boxes. This enables the use of a tableaux-based reasoner for complete T-Box inference. Consequently, inference over the A-Box is our focus, and we term this **type inference** [MRS12], which is to derive for each instance its membership of classes

and properties. We also restrict our approach to only consider updates to the A-Box, and assume the T-Box is static.

By applying a tableaux-based reasoner for T-Box classification, we translate the the classified T-Box into an **inference framework**, which is used for computing and materialising the results of type inference, where queries only need to read the materialisation. The inference framework contains **transformation rules** handling OWL 2 constructors; in particular, they not only cover all the **OWL 2 RL/RDF rules** [Krö12a] (except where the **Unique Name Assumption** (**UNA**) conflicts, as our approach follows the UNA while OWL does not), but also handle some extra axioms beyond OWL 2 RL for a more complete inference. Our approach then acts like a compiler, which is able to implement the inference framework in many mainstream database systems, such as a **Relational Database Management System** (**RDBMS**) (e.g. Postgres [Mom01] or Microsoft SQL Server [DB87]) or a **distributed Big Data system** (e.g. Spark [KKWZ15] or MapReduce [DG08] running over a **Hadoop Distributed File System** (**HDFS**) [Whi15]).

When the A-Box size is small enough to be stored inside an RDBMS, the inference framework is implemented as an **Auto Type Inference Database** (**ATIDB**) containing tables and triggers, and we call this part of work **SQOWL2** [LM13]. When inserts or deletes of A-Box facts are executed in database transactions over the ATIDB, triggers acting actively as **Event Condition Action** (**ECA**) rules, are automatically invoked by these updates to analyse how the previous type inference materialisation should be **incrementally** updated. In a nutshell, triggers invoked by data inserts materialise derivations computed from the newly inserted data and the existing materialisation. However, performing type inference from data deletes is significantly more difficult than handling data inserts, and triggers invoked by deletes conduct what we call a **label & check** process [LM15] (a variant of the **Delete & Rederive** (**DRed**) algorithm [GMS93] that can be implemented in an RDBMS). The label & check process first labels all data that might be affected because of the deletes, and then removes the data that has been labelled and can not be re-inferred from non-labelled data.

Incremental inference avoids the unnecessary re-computation of the whole inference mate-

rialisation required by non-incremental reasoners (e.g. Oracle's RDF Store, the Lite version of OWLim) or partial-incremental reasoners (e.g. WebPIE, which only performs incremental inference for dealing with inserts but not deletes). Moreover, the label & check process is able to handle deletes over recursive views that can be defined in OWL 2 RL ontologies. Also, inference by the ATIDB is available as atomic consequences of data inserts or deletes, which preserves the **Atomicity, Consistency, Isolation, Durability** (**ACID**) properties [HR83] of database transactions. Such inference is known as **transactional reasoning** [MRS12], which guarantees that the ATIDB can never enter an inconsistent state. However, most reasoners (e.g. Minerva and Oracle's RDF Store) fail to provide transactional reasoning, even though they store the inference inside an RDBMS, as they require a separate inference process which is often performed outside an RDBMS.

When the A-Box size is too large to be stored in an RDBMS, our inference framework can be implemented as Spark programmes, which perform type inference for OWL 2 RL ontologies in an HDFS, and we name this part of work **SPOWL**. Spark programmes are directly translated from a classified T-Box, i.e. only axioms contained in the T-Box are compiled into Spark programmes. This avoids unnecessary rule-matching faced by most large reasoners (e.g. WebPIE, Cichlid [GWW+15]), which simply evaluate a set of entailment rules no matter whether these rules are relevant to the input ontology or not. Spark programmes act as non-active **if-then** rules, and are able to compute and materialise results of type inference in a scalable manner. The materialisation is computed by iteratively executing the Spark programmes until no new inference can be derived. The iterative execution follows an optimised order based on the bottom-up hierarchy of a T-Box, so that the number of iterations can be reduced.

From the viewpoint of implementation, SQOWL2 gathers all trigger fragments associated with each table and implements them as a single trigger, which significantly improves the performance of type inference compared to SQOWL, which implements the trigger fragments separately. SQOWL2 also provides some optimisations, such as adding indexes and **Foreign Keys** (**FKs**) or tuning the ATIDB to use a more efficient execution plan, which improve its performance of not only inference computation but also query processing. For SPOWL, it inherits the light and fast data processing from the Spark framework, which caches data in distributed

memory as much as possible, and schedules jobs in a more flexible and parallelised manner than the sequential job scheduling of MapReduce. In a nutshell, SPOWL caches frequently used datasets to avoid unnecessary re-computation, and for joining a large set of data with a small set, it partitions the large dataset and copies the small set to each of the partitions, which reduces the amount of data requiring to be shuffled.

We have also evaluated our work over the well-known **Lehigh University Benchmark** (**LUBM**) [GPH05]. The completeness of our approach was only experimentally evaluated, and the results show both implementations are able to completely process the 14 LUBM queries w.r.t. LUBM T-Box and any arbitrary LUBM A-Box. This is often failed to be achieved by most state-of-the-art rule-based reasoners [SGH10]. SQOWL2's performance of processing data inserts, data deletes and queries was further compared to a non-materialising system Stardog and a materialisation-based system OWLim. Results show that SQOWL2, when compared to Stardog and OWLim, provides faster query processing than the two comparison systems, at the expense of slower data updating. With regard to SPOWL, its scalability for handling ontologies beyond the capability of an RDBMS was evaluated on a small distributed cluster of 9 nodes running on a cloud computing platform[4]. The inference materialising and query processing were both scaled up to run over approximately 270 million A-Box facts.

1.2 Contributions

The primary contribution of the thesis is that from the previous work SQOWL [MRS09, MRS10], we extend its inference framework and implement the extended framework as two new systems. The extensions can be summarised as follows:

1. SQOWL2 extends SQOWL, which only handles data inserts, to now also support type inference from data deletes in an RDBMS. In order to handle data deletes, we modify the well-known DRed algorithm to assign every piece of data stored in an ATIDB a state. This state of data allows DRed's delete & rederive process to be changed to SQOWL2's

[4] `https://www.doc.ic.ac.uk/csg/services/cloud`

more efficient label & check process, which updates the inference materialisation from data deletes in a transactional and incremental manner. The work on SQOWL2 for handling data updates has been published in [LM15, LM16].

2. SQOWL2 supports type inference over the ontologies expressed in OWL 2 as compared to SQOWL, which supports OWL 1. In particular, SQOWL2 handles every OWL 2 constructor by transforming their constructed axioms into triggers, which surpass the OWL 2 RL/RDF rules. SQOWL2 is conjectured to be a sound and complete implementation of the OWL 2 RL/RDF rules for type inference over OWL 2 RL ontologies. The work in this part of the thesis has been published in [LM13, LM14].

3. SPOWL further extends SQOWL2 to support type inference over ontologies with large A-Boxes that are not small enough for an RDBMS to handle. SPOWL translates OWL 2 axioms into Spark programmes, which compute and materialise the results of type inference in a Big Data system. In particular, we introduce a new technique of compiling an OWL ontology directly into Spark programmes that implement the ontology axioms. SPOWL focuses on scalable type inference for OWL 2 RL ontologies, and thus it sacrifices the transactional and incremental properties held by SQOWL2.

1.3 Thesis Outline

In the next chapter we provide some background on OWL 2 ontologies; in particular, we outline the T-Box and A-Box of an ontology, illustrate some simple examples of inference, and we then introduce in detail how the Direct Semantics interprets every OWL 2 expression and defines the satisfying conditions for every OWL 2 axiom.

Chapter 3 reviews in detail the inference in OWL 2 as compared to Chapter 2 which only outlines some inference examples. We first provide a list of inference tasks often considered by reasoners, and some properties for evaluating a reasoner. Then, we move to a thorough illustration of tableaux-based inference and rule-based inference (i.e. query-rewriting and materialised approaches), and also analyse the advantages and disadvantages of these inference techniques.

Chapter 4 covers the first part of SQOWL2, which performs type inference for OWL 2 from only data inserts. We start by a motivating example to show some problems that might occur in type inference, and then move to the details of how an ATIDB is established to perform type inference from data inserts. In particular, we illustrate a canonical schema for representing OWL classes and properties, where triggers can be created. We show in detail how all OWL 2 constructors are handled by triggers, and summarise some optimisations that can be used to further improve SQOWL2.

Chapter 5 focuses on how SQOWL2 performs type inference from data deletes. Like the previous chapter, we first use a motivating example to address the problem of handling data deletes, and we also review the DRed algorithm, on which our approach is based. Then, we introduce a state which we assign to each data item stored in ATIDB, and how the state can be changed by database updates, which we group as ontology and reasoner updates. Afterwards, the label & check process is defined in detail by illustrating how this process can be used to solve the problem addressed in the motivating example. We also illustrate how triggers in Chapter 4 are extended to fully support both data inserts and deletes. Finally, we analyse SQOWL2 on its soundness and completeness of type inference.

Chapter 6 changes the attention of type inference from an RDBMS to a Big Data system. We first show the issue of simply evaluating entailment rules for inference materialisation. After briefly reviewing the Spark, we outline the architecture of SPOWL, and illustrate the operation of SPOWL by a small example. Then, a detailed transformation from OWL 2 RL axioms to Spark programmes is provided. Finally, we illustrate how the Spark programmes can be executed in an optimised order following the bottom-up hierarchy of a T-Box.

Note that in order to describe SQOWL2 and SPOWL in an implementation-independent manner, the triggers and Spark programmes are all presented in a logical manner in Chapters 4, 5 and 6. The implementation details are provided in Chapter 7. In particular, We show how physical triggers are created and operated in Transact SQL [KGZ99], and how logical Spark programmes are implemented into real code. Evaluating the implementations is the second part of this chapter, we show how experiments are designed and conducted, and provide detailed

analysis of the results.

Finally, Chapter 8 draws some conclusions and discusses future work.

Chapter 2

Review of Web Ontology Language

2.1 Introduction

In Chapter 1, we have already used the OWL to express certain ontological information, such as (1.1) – (1.7). As a state-of-the-art KR language, OWL supplies a formalised way of describing knowledge as **ontologies**, in a form that is accessible to and processable by machines. The most recent release, OWL 2, provides two formal semantics, **Direct Semantics** and **RDF-based Semantics** [Sch12], which interpret components in ontologies. Based on the interpretations, **inference** (*a.k.a.* **reasoning**) can be performed to derive implicit consequences from the knowledge explicitly represented. This thesis focuses on the Direct Semantics, as its underpinning $\mathcal{SROIQ}$ DL is decidable, while the RDF-based Semantics, which is an extension of D-Entailment [HPS14], has undecidable reasoning [Mot07].

In this chapter, Section 2.2 first provides an overview of how OWL 2 can be used to express knowledge, and illustrates two important inference tasks (a detailed review of inference techniques and systems is given in Chapter 3). Next, Section 2.3 details the interpretation for OWL 2 expressions, and satisfying conditions for OWL 2 axioms as defined in the Direct Semantics. Throughout this chapter, we continue to use the beer ontology in examples to illustrate certain concepts.

2.2 OWL 2 Overview

An OWL 2 ontology expresses knowledge schema in a T-Box and known facts in an A-Box.

2.2.1 Terminological Box

In a T-Box of an ontology, a **class**, such as CzechLager, Lager and CzechBeer in (1.1), can be characterised as a unary relation used for denoting a set of **individuals**, such as CzechPaleLager and CzechDarkLager in (1.2) – (1.4). In the beer ontology, we may specify another class Ale containing warm-fermented beer individuals, such as BritishBrownAle and EnglishPorter, and two additional classes Flavour and Colour, which respectively include beer flavours (e.g. Bitter, Malty) and beer colours (e.g. Dark, Brown and Pale).

A **property** in an ontology denotes a binary relation, and it can be an **object property** or a **data property**. An object property relates an individual to another individual, while a data property relates an individual to a data value. For example, we may specify an object property hasFlavour to relate beers to their flavours (e.g. CzechPaleLager has a Bitter flavour), and a data property hasAlcoholLevel to relate beers to their alcohol percentage (e.g. EnglishPorter has 5.4% alcohol by volume).

OWL 2 also provides numerous **constructors** (e.g. SubClassOf, PropertyDomain) to express more complex relations or constraints, which are more naturally called **axioms**. For example, a subsumption relation between two classes (e.g. axioms (1.5) and (1.6)) can be specified by a constructor SubClassOf (symbolised as $\sqsubseteq$ in the DL syntax[1]), which means all individuals in the subclass are also instances of the super class. In the beer ontology, we may further specify PaleAle and Porter as two subclasses of Ale by axioms (2.1) and (2.2) in the DL syntax.

$$\mathsf{PaleAle} \sqsubseteq \mathsf{Ale} \tag{2.1}$$

$$\mathsf{Porter} \sqsubseteq \mathsf{Ale} \tag{2.2}$$

Specifying other relations, such as equivalence between classes or properties (using the constructor EquivalentClasses or EquivalentProperties), and constraints, such as universal and ex-

[1] In the thesis, we follow the DL syntax because of its compact format of specifying the formal structure of ontologies.

istential restrictions (using constructors AllValuesFrom and SomeValuesFrom) are described in Section 2.3, where a thorough description of the Direct Semantics is given.

Note that, we denote classes by convention beginning with upper case letters and in a brown colour, properties by convention starting with lower case letters and in a green colour, and finally individuals starting with upper case and in a blue colour. Moreover, unless specified, the term **property** will refer to an object property, as very few axioms can be specified on a data property.

2.2.2 Assertional Box

The A-Box of an ontology is used for storing known facts (*a.k.a.* data) in a knowledge domain as **assertions** (or **facts**). A **class fact** (using the constructor ClassAssertion) specifies for each individual its membership of a class, such as specifying CzechPaleLager as an instance of Lager. A **property fact** (using the constructor PropertyAssertion) defines for a pair of individuals its membership of a property, such as specifying ⟨CzechPaleLager, Bitter⟩ as an instance of hasFlavour. Moreover, the individuals that are related by a property are often called **subjects** of this property, such as CzechPaleLager of hasFlavour. The individuals which a property relates the subjects to are called **objects**, such as Bitter of hasFlavour.

In DL, we use the syntax of $C(a)$ for class facts, and the syntax of $P(a, b)$ for property facts. The facts we have mentioned for the beer ontology so far are summarised as follows (where (2.10) and (2.11) are two property facts, others are class facts):

Lager(CzechPaleLager)	(1.2)	Flavour(Malty)	(2.6)
CzechBeer(CzechPaleLager)	(1.3)	Colour(Dark)	(2.7)
CzechLager(CzechDarkLager)	(1.4)	Colour(Brown)	(2.8)
Ale(BritishBrownAle)	(2.3)	Colour(Pale)	(2.9)
Ale(EnglishPorter)	(2.4)	hasFlavour(CzechPaleLager, Bitter)	(2.10)
Flavour(Bitter)	(2.5)	hasAlcoholLevel(EnglishPorter, 5.4%)	(2.11)

Note that in most ontologies, such as DBpedia and WordNet, the size of their T-Boxes are

often quite small as compared with their A-Boxes, which are normally very large (e.g. billions of facts). We may extend the beer ontology with more facts as follows:

$$\mathsf{PaleAle}(\mathsf{BritishGoldenAle}) \quad (2.12)$$
$$\mathsf{PaleAle}(\mathsf{CreamAle}) \quad (2.13)$$
$$\mathsf{PaleAle}(\mathsf{BlondeAle}) \quad (2.14)$$
$$\mathsf{Porter}(\mathsf{BalticPorter}) \quad (2.15)$$
$$\mathsf{Porter}(\mathsf{EnglishPorter}) \quad (2.16)$$
$$\mathsf{Porter}(\mathsf{AmericanPorter}) \quad (2.17)$$
$$\mathsf{Flavour}(\mathsf{Sweet}) \quad (2.18)$$
$$\mathsf{Colour}(\mathsf{Amber}) \quad (2.19)$$
$$\mathsf{hasFlavour}(\mathsf{EnglishPorter}, \mathsf{Malty}) \quad (2.20)$$
$$\mathsf{hasFlavour}(\mathsf{BritishGoldenAle}, \mathsf{Bitter}) \quad (2.21)$$
$$\mathsf{hasColour}(\mathsf{EnglishPorter}, \mathsf{Dark}) \quad (2.22)$$
$$\mathsf{hasColour}(\mathsf{BalticPorter}, \mathsf{Dark}) \quad (2.23)$$

As asserted by (2.12) – (2.14), we introduce three new pale ales: BritishGoldenAle, CreamAle and BlondeAle, and by (2.15) – (2.17), we have three new porters: BalticPorter, EnglishPorter and AmericanPorter. In addition, by (2.18) Flavour is extended with one more individual Sweet, and Colour includes Amber by (2.19). Moreover, property facts (2.20) and (2.21) respectively specify the flavours of EnglishPorter and BritishGoldenAle, and (2.22) and (2.23) describe the colours of EnglishPorter and BalticPorter, respectively.

2.2.3 Inference

Inference (*a.k.a.* **reasoning**[2]) can be described as the process of deriving implicit information from the explicitly specified information, based on the Direct Semantics. It is often considered as the key difference between knowledge modelling and traditional conceptual modelling such as the relational and ER models. Inference considered in OWL 2 can be categorised as **T-Box inference** and **A-Box inference**; the former focuses on inference tasks over the knowledge schema, and the latter addresses inference problems over the ontology data.

In both inference categories, there are more detailed tasks, such as **classification** in T-Box inference, and **instance checking** in A-Box inference. A software system which solves all or some of these inference tasks is called a **reasoner**. To solve a particular inference task, different inference methods can be adopted by a reasoner, and two of them, **tableaux-based inference**

[2]We use the two terms interchangeably through this thesis.

and **rule-based inference**, are considered as state-of-the-art. The former is used for complete inference, while the latter is often applied for inference over large A-Boxes.

Here, we give a brief overview of two important inference tasks (i.e. classification and instance checking), a complete review of all inference tasks and a detailed description about the two inference methods are provided in Chapter 3.

Classification in T-Box Inference

An important task to consider in T-Box inference is to classify all classes in an ontology, such that all subsumption relations between classes are identified, so that a class hierarchy can be established. For example, in the beer ontology, we shall have a class Beer, and Ale and Lager are two of its subclasses as stated by (2.24) and (2.25) below:

$$\mathsf{Ale} \sqsubseteq \mathsf{Beer} \qquad (2.24) \qquad\qquad \mathsf{Lager} \sqsubseteq \mathsf{Beer} \qquad (2.25)$$

If we take the two subclasses PaleAle and Porter of Ale in (2.1) and (2.2) into account, using the transitivity of subsumption relations (i.e. if $C \sqsubseteq D$ and $D \sqsubseteq E$, then $C \sqsubseteq E$), we can infer two new subsumption relations from PaleAle to Beer in (2.26), and from Porter to Beer in (2.27).

$$\mathsf{PaleAle} \sqsubseteq \mathsf{Beer} \qquad (2.26) \qquad\qquad \mathsf{Porter} \sqsubseteq \mathsf{Beer} \qquad (2.27)$$

Instance Checking in A-Box Inference

Instance checking is an important task in A-Box inference, which checks for whether an individual belongs to a class or not. For example, using the subsumption relation from PaleAle to Ale in (2.1), we can verify BritishGoldenAle is an Ale from the A-Box fact (2.12). Indeed, because of (2.1), CreamAle and BlondeAle, which are instances of PaleAle specified by (2.13) and (2.14), can also be verified as members of Ale.

Note that instance checking might result in a third status unknown besides true and false, because OWL 2 follows an **open-world assumption** (**OWA**) [Min82] rather than a **closed-world assumption** (**CWA**) [Rei78]. The possibility of an A-Box fact holding is preserved in the OWA, even if the fact is not recorded, i.e. verifying any unpersisted fact always results in

unknown, whilst in the CWA it would result in false. For example, in the beer ontology (as we have introduced so far), when checking in the OWA if CreamAle is a Lager, we are unable to give a true or false value. Hence, we say the fact of Lager(CreamAle) is unknown. OWA contradicts the CWA (commonly adopted in the field of databases), which assumes that any fact not recorded should always be evaluated to false. In fact, the answer false is relatively uncommon in an OWA database, but we will show one example later when looking at DisjointClasses.

Checking whether every individual belongs to a particular class can be extremely inefficient, when the size of A-Box is large. Therefore, based on instance checking, another inference task, called **instance retrieval** is often used when querying all individuals of a class. Instance retrieval computes all individuals of a particular class. For example, we have already shown the class Ale so far includes BritishBrownAle, BritishGoldenAle, CreamAle, BlondeAle, BalticPorter, EnglishPorter and AmericanPorter as its instances. Except BritishBrownAle and EnglishPorter, which are explicitly asserted as instances of Ale, others are implicitly derived. EnglishPorter as a member of Ale is a special case, as this fact is not only explicitly specified by (2.4), but also can be implicitly inferred by (2.2) and (2.16). For this case, even if we remove the explicit fact (2.4), this fact still holds implicitly.

2.3 The Direct Semantics of OWL 2

As clarified in Section 2.1, the semantics considered in this thesis is the Direct Semantics (*a.k.a.* OWL 2 DL), which is based on the $\mathcal{SROIQ}$ DL (a decidable fragment of the FOL). To review OWL 2 DL, we first outline in Section 2.3.1 its **interpretation** $\mathcal{I}$, which interprets OWL 2 expressions, and defines the conditions for satisfying OWL 2 axioms. Then, Section 2.3.2 and Section 2.3.3 complete $\mathcal{I}$ for all semantics related to classes and properties, respectively.

2.3.1 Interpretation $\mathcal{I}$ of OWL 2 DL

Interpreting OWL 2 expressions

For the three basic ontology components, i.e. classes, properties and individuals, the interpretation $\mathcal{I}$ defines them in a set-theoretic way, as shown in Definition 2.1:

Definition 2.1 *An interpretation $\mathcal{I}$ of OWL 2 DL consists of a non-empty set of objects Δ (a.k.a. the knowledge universe), and we may use the interpretation as a function on classes, properties and individuals to assign them some set-theoretic elements of Δ (denoted as $a^{\mathcal{I}}$, $C^{\mathcal{I}}$ and $P^{\mathcal{I}}$, respectively). In particular, $\mathcal{I}$ associates:*

- *each individual a with an object in the universe*[3]*: $a^{\mathcal{I}} \in \Delta$*
- *each class C with a set of objects: $C^{\mathcal{I}} \subseteq \Delta$*
- *each property P with a subset of the Cartesian product of the universe: $P^{\mathcal{I}} \subseteq \Delta \times \Delta$*

OWL 2 expressions can be **atomic** (i.e. OWL classes and properties), or **complex** (i.e. expression formed by combining some basic OWL components and constructors). We denote by C, D and E for atomic classes, and by P, Q and R for atomic properties, and by C_E and P_E to refer general expressions (either atomic or complex).

The expression of $C \sqcap D$ (e.g. Lager $\sqcap$ CzechBeer in (1.1)) is a complex expression formed by a constructor IntersectionOf (symbolised as $\sqcap$ in DL). $C \sqcap D$ expresses that the same individuals exist in both C and D, and it is interpreted by $\mathcal{I}$ as $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$. Because $C \sqcap D$ denotes a set of individuals, we call it a **class expression**, while a **property expression** denotes pairs of individuals. The constructor InverseOf is used to form a (complex) property expression denoted as P^{-}, which contains inverse pairs of individuals of P, and it is interpreted by $\mathcal{I}$ as $(P^{-})^{\mathcal{I}} = \{\langle y, x\rangle \mid \langle x, y\rangle \in P^{\mathcal{I}}\}$.

[3]Remember that OWL 2 does not follow UNA, and indeed by this definition, two individuals may be assigned to the same object (i.e. $a^{\mathcal{I}} = b^{\mathcal{I}}$).

Satisfying T-Box axioms

Besides interpreting expressions, $\mathcal{I}$ defines the truth conditions on which OWL 2 axioms hold. We denote by $\mathcal{I} \models axiom$ for the situation that the *axiom* holds on some conditions defined in $\mathcal{I}$ (i.e. $\mathcal{I}$ satisfies the axiom). For example, the condition on which $\mathcal{I}$ satisfies (2.1), a subsumption relation from PaleAle to Ale, is shown as:

$$\mathcal{I} \models \mathsf{PaleAle} \sqsubseteq \mathsf{Ale} \text{ iff } \mathsf{PaleAle}^{\mathcal{I}} \subseteq \mathsf{Ale}^{\mathcal{I}}$$

which means that $\mathcal{I}$ satisfies (2.1) if and only if the objects which $\mathcal{I}$ assigns to PaleAle is a subset of the objects that $\mathcal{I}$ associates with Ale (denoted by $\subseteq$).

Satisfying A-Box facts

$\mathcal{I}$ also defines the truth condition for satisfying A-Box facts. Satisfying a class fact by $\mathcal{I}$ is denoted as $\mathcal{I} \models C(a)$ iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$. For example, $\mathcal{I}$ satisfying that the individual BritishBrownAle as an instance of Ale (i.e. (2.3)) can be written as:

$$\mathcal{I} \models \mathsf{Ale}(\mathsf{BritishBrownAle}) \text{ iff } \mathsf{BritishBrownAle}^{\mathcal{I}} \in \mathsf{Ale}^{\mathcal{I}}$$

As can be seen, holding a class fact requires that the object assigned to the individual belongs to the objects assigned to the class (denoted by $\in$). Similarly, satisfying a property fact is written as $\mathcal{I} \models P(a, b)$ iff $\langle a^{\mathcal{I}}, b^{\mathcal{I}} \rangle \in P^{\mathcal{I}}$. For example, satisfying the property fact EnglishPorter has a Malty flavour defined by (2.20) can be illustrated as:

$$\mathcal{I} \models \mathsf{hasFlavour}(\mathsf{EnglishPorter}, \mathsf{Malty}) \text{ iff } \langle \mathsf{EnglishPorter}^{\mathcal{I}}, \mathsf{Malty}^{\mathcal{I}} \rangle \in \mathsf{hasFlavour}^{\mathcal{I}}$$

2.3.2 Class-related Semantics

As illustrated in Section 2.3.1, OWL 2 expressions (either complex or atomic) can be used to represent a set of individuals (i.e. class expressions C_E interpreted by $\mathcal{I}$ as ${C_E}^{\mathcal{I}} \subseteq \Delta$) or to denote a set of individual pairs (i.e. property expressions P_E interpreted as ${P_E}^{\mathcal{I}} \subseteq \Delta \times \Delta$). Then, axioms can be constructed by expressing either a subsumption or an equivalence relation between two expressions (either between two class expressions or between two property

expressions). In this section, we focus on class expressions and related axioms, and leave property expressions and related axioms in Section 2.3.3.

SubClassOf and EquivalentClasses:

As we have already exemplified in Section 2.2.1, the SubClassOf constructor is used to express a subsumption relation between two atomic classes C and D (e.g. (2.1) and (2.2)). In fact $C \sqsubseteq D$ can be generalised as $C_{E_1} \sqsubseteq C_{E_2}$, where C_{E_1} and C_{E_2} can be either atomic or complex, and we call C_{E_1} the subclass expression, and C_{E_2} the super-class expression. To satisfy a SubClassOf axiom, we need to ensure the set of individuals assigned to the subclass expression is a subset of the set of individuals assigned to the super-class expression, which is defined by $\mathcal{I}$ as follows:

$$\mathcal{I} \models C_{E_1} \sqsubseteq C_{E_2} \text{ iff } {C_{E_1}}^{\mathcal{I}} \subseteq {C_{E_2}}^{\mathcal{I}}$$

An equivalence relation between two class expressions can be formed by another constructor EquivalentClasses (with the symbol $\equiv$), in order to express that two class expressions specify exactly the same set of individuals. For example, we may add to the beer ontology a new class LiquidBread, which is equivalent to the class Beer:

$$\mathsf{Beer} \equiv \mathsf{LiquidBread} \tag{2.28}$$

The truth condition for satisfying $C_{E_1} \equiv C_{E_2}$ is defined as:

$$\mathcal{I} \models C_{E_1} \equiv C_{E_2} \text{ iff } {C_{E_1}}^{\mathcal{I}} = {C_{E_2}}^{\mathcal{I}}$$

which means that the set of objects which $\mathcal{I}$ assigns to C_{E_1} is exactly the same set of objects assigned to C_{E_2}. Note that the semantics of an EquivalentClasses axiom $C_{E_1} \equiv C_{E_2}$ is logically equivalent to two SubClassOf axioms $C_{E_1} \sqsubseteq C_{E_2}$ and $C_{E_2} \sqsubseteq C_{E_1}$, i.e.:

$$\mathcal{I} \models C_{E_1} \equiv C_{E_2} \text{ iff } \mathcal{I} \models C_{E_1} \sqsubseteq C_{E_2} \text{ and } \mathcal{I} \models C_{E_2} \sqsubseteq C_{E_1}$$

IntersectionOf, UnionOf and ComplementOf:

The constructor IntersectionOf can specify a complex class expression denoting common individuals from two class expressions (denoted as $C_{E_1} \sqcap C_{E_2}$, where C_{E_1} and C_{E_2} can be atomic or complex). $\mathcal{I}$ interprets $C_{E_1} \sqcap C_{E_2}$ as the objects which $\mathcal{I}$ assigns to both C_{E_1} and C_{E_2}:

$$(C_{E_1} \sqcap C_{E_2})^{\mathcal{I}} = {C_{E_1}}^{\mathcal{I}} \cap {C_{E_2}}^{\mathcal{I}}$$

Besides IntersectionOf which specifies the *and* logic in the set theory, OWL 2 also provides UnionOf and ComplementOf to express logical *or* and *not*, respectively. Expressions using UnionOf are denoted as $C_{E_1} \sqcup C_{E_2}$, which represents all elements which belong to C_{E_1} or C_{E_2}, and expressions using ComplementOf are denoted as $\neg C_E$, which specifies the set of elements that are not instances of C_E. $\mathcal{I}$ interprets the two types of expressions as:

$$(C_{E_1} \sqcup C_{E_2})^{\mathcal{I}} = {C_{E_1}}^{\mathcal{I}} \cup {C_{E_2}}^{\mathcal{I}}$$

$$(\neg C_E)^{\mathcal{I}} = \Delta \backslash {C_E}^{\mathcal{I}}$$

Considering the beer ontology, we may use Ale $\sqcap$ Lager to specify the beers which are both an ale and a lager, Ale $\sqcup$ Lager to express the beers which are an ale or a lager, and $\neg$Ale to denote all individuals that are not ales – might not be a beer ($\neg$Ale $\sqcap$ Beer is the proper expression to denote beers which are not ales). $\mathcal{I}$ interprets them as follows:

$$(\mathsf{Ale} \sqcap \mathsf{Lager})^{\mathcal{I}} = \mathsf{Ale}^{\mathcal{I}} \cap \mathsf{Lager}^{\mathcal{I}}$$

$$(\mathsf{Ale} \sqcup \mathsf{Lager})^{\mathcal{I}} = \mathsf{Ale}^{\mathcal{I}} \cup \mathsf{Lager}^{\mathcal{I}}$$

$$(\neg\mathsf{Ale})^{\mathcal{I}} = \Delta \backslash \mathsf{Ale}^{\mathcal{I}}$$

DisjointClasses **and** DisjointUnion**:**

By using the expressions formed by IntersectionOf, UnionOf and ComplementOf, we are able to express some axioms like DisjointClasses and DisjointUnion. DisjointClasses is used to describe that several class expressions are disjoint with each other (i.e. they do not share the same individuals, and thus the intersection of these class expressions should be empty). To denote an empty set, OWL 2 provides a special class called Nothing (symbolised as $\bot$ in DL), which does not contain any individual. The opposite of an empty set is the *universe* containing everything of a knowledge domain, which is represented by another class Thing denoted as $\top$ in DL. $\mathcal{I}$ interprets Thing and Nothing as:

$$\mathsf{Thing}^{\mathcal{I}} = \Delta$$

$$\mathsf{Nothing}^{\mathcal{I}} = \emptyset$$

Thus, specifying several classes are disjoint can be made by setting the intersection of them to a subclass of Nothing. For example, in the beer ontology, we may restrict that ales and lagers are two different beers, and therefore classes Ale and Lager should not contain the same beer individuals, which can be specified by (2.29).

$$\mathsf{Ale} \sqcap \mathsf{Lager} \sqsubseteq \bot \tag{2.29}$$

The truth condition for holding such an axiom is defined by $\mathcal{I}$ as follows:

$$\mathcal{I} \models \mathsf{Ale} \sqcap \mathsf{Lager} \sqsubseteq \bot \text{ iff } \mathsf{Ale}^{\mathcal{I}} \cap \mathsf{Lager}^{\mathcal{I}} = \emptyset$$

In DL, there might exist more than one way to specify the same type of axioms. For example, (2.29) can also be expressed as (2.30), which uses the constructor ComplementOf to make Lager a subset of the complement of Ale.

$$\mathsf{Lager} \sqsubseteq \Delta \backslash \mathsf{Ale} \tag{2.30}$$

Note that the constructor DisjointClasses is not limited to specify disjointness between only two class expressions, but a finite number of class expressions. However, using the DL syntax like (2.29) or (2.30) to specify disjointness between every pair of n class expressions will require $(n-1)!$ DL statements contained in an ontology. To mitigate this, a more compact format, written as $\mathsf{DisCla}(C_{E_1}, \ldots, C_{E_n})$, is offered by DL, and its satisfying condition is defined as:

$$\mathcal{I} \models \mathsf{DisCla}(C_{E_1}, \ldots, C_{E_n}) \text{ iff } {C_{E_i}}^{\mathcal{I}} \cap {C_{E_j}}^{\mathcal{I}} = \emptyset, \text{where } 1 \leq i < j \leq n$$

OWL 2 also provides another constructor DisjointUnion, which can be used to define an atomic class as a union of some disjoint class expressions. DisjointUnion is actually a syntactic sugar combining DisjointClasses and UnionOf. For instance, we can define Beer as a union of the two disjoint classes Ale and Lager by adding an extra axiom (2.31) alongside (2.29) or (2.30).

$$\mathsf{Beer} \equiv \mathsf{Ale} \sqcup \mathsf{Lager} \tag{2.31}$$

A more compact syntax for representing DisjointUnion is $\mathsf{DisUni}(C, C_{E_1}, \ldots, C_{E_n})$ (i.e. C as a union of $C_{E_1}, \ldots, C_{E_n}$ which are pairwise disjoint). Thus, to make the beer ontology more concise, (2.29) and (2.31) can be merged as:

$$\mathsf{DisUni}(\mathsf{Beer}, \mathsf{Ale}, \mathsf{Lager}) \tag{2.32}$$

Recall the aforementioned instance checking question in Section 2.2.3: whether CreamAle be-

longs to Lager or not? This question was evaluated to unknown, as OWL 2 follows the OWA; however, because we have added some extra statements, the answer to this question might need to be reconsidered. Firstly, (2.1) and (2.24) together denote a subsumption hierarchy, PaleAle $\sqsubseteq$ Ale $\sqsubseteq$ Beer (i.e. every instance of PaleAle is an Ale and also a Beer). Hence, we can easily derive that CreamAle is both an Ale and a Beer, because CreamAle has been asserted as a PaleAle by (2.13). Moreover, the definition of Beer by (2.32) restricts that a Beer can only be either an Ale or a Lager, which implies that CreamAle cannot be a Lager because it is already inferred an Ale. Therefore, the fact of Lager(CreamAle) is false now.

The above discussion also highlights the difference between OWA and CWA when modelling an information domain [GM05]; in OWA, we start to model a knowledge domain from the case that everything is possible, and then restrict the model by adding more relations and constraints. However, in CWA, we follow the principle that everything is not true unless we explicitly store them, then we build up the model by adding more facts.

AllValuesFrom, SomeValuesFrom, HasValue and OneOf:

OWL 2 offers constructors AllValuesFrom (symbolised as $\forall$) and SomeValuesFrom (symbolised as $\exists$) for specifying universal and existential quantification, respectively. AllValuesFrom can be used to form a class expression $\forall P.C_E$, which defines a set of individuals x such that if x is related to y by P, then y must be inferred as a member of C_E. This is interpreted by $\mathcal{I}$ as:

$$(\forall P.C_E)^{\mathcal{I}} = \{x \mid \forall y : \text{if} \langle x, y \rangle \in P^{\mathcal{I}} \text{ then } y \in {C_E}^{\mathcal{I}}\}$$

Note that as a result of this interpretation, if there is no pair $\langle x, y \rangle$ in P, the expression $\forall P.C_E$ still holds for x.

In the beer ontology, we may use $\forall$hasFlavour.Flavour to specify that any individual which hasFlavour relates to is a Flavour, and use $\forall$hasColour.Colour to specify that any individual which hasColour relates to is a Colour. Obviously, this is generally true for all objects in the knowledge domain of beer; therefore, we can specify the universe class Thing as a subclass of $\forall$hasFlavour.Flavour by (2.33), and as a subclass of $\forall$hasColour.Colour by (2.34), in order to apply the two universal quantifications to every individual in the ontology.

$$\top \sqsubseteq \forall \mathsf{hasFlavour}.\mathsf{Flavour} \quad (2.33) \qquad \top \sqsubseteq \forall \mathsf{hasColour}.\mathsf{Colour} \quad (2.34)$$

The axioms (2.33) and (2.34) actually specify the PropertyRange of hasFlavour and hasColour, respectively. Indeed, PropertyDomain and PropertyRange axioms, which we detail the semantics later in Section 2.3.3, adopt AllValuesFrom expressions to respectively restrict property subjects and objects to individuals from particular classes.

An expression formed by SomeValuesFrom specifies an existential quantification. Its DL syntax $\exists P.C_E$ defines a set of individuals x such that there exists at least one pair of individuals $\langle x, y\rangle$ in P, where the individual y comes from the class expression C_E. $\mathcal{I}$ interprets $\exists P.C_E$ as:

$$(\exists P.C_E)^{\mathcal{I}} = \{x \mid \exists y : \langle x, y\rangle \in P^{\mathcal{I}} \text{ and } y \in {C_E}^{\mathcal{I}}\}$$

We may set the C_E of $\exists P.C_E$ to be the universe class Thing, i.e. $\exists P.\top$. We call $\exists P.C_E$ in the case of $C_E \equiv \top$ **unqualified** and the expression can be simplified as $\exists P$; otherwise, if C_E denotes a class expression other than Thing, we say the expression is **qualified**. Note that the unqualified case $\forall P$ for AllValuesFrom expression $\forall P.C_E$, just denotes $\top$. This can be explained by the interpretation of $\forall P$:

$$(\forall P)^{\mathcal{I}} = \{x \mid \forall y : \text{if} \langle x, y\rangle \in P^{\mathcal{I}} \text{ then } y \in \Delta\}$$

which means that $\forall P$ includes:

1. all individuals x which are related by P (because $y \in \Delta$ is always true);
2. all individuals other than x (since $\forall P$ still holds for individuals which are not recorded as related by P because of the definition of an AllValuesFrom expression).

Thus, $\forall P$ is further interpreted as $(\forall P)^{\mathcal{I}} = \Delta$.

OWL 2 allows for specifying an atomic class as a subset of, or as equivalent to, a SomeValuesFrom expression (i.e. $C \sqsubseteq \exists P.C_E$ or $C \equiv \exists P.C_E$). For example, in the beer ontology, we may restrict that each individual in Beer must have at least one flavour by:

$$\mathsf{Beer} \sqsubseteq \exists \mathsf{hasFlavour}.\mathsf{Flavour} \quad (2.35)$$

This axiom is satisfied if the following condition holds:

$$\mathcal{I} \models \mathsf{Beer} \sqsubseteq \exists \mathsf{hasFlavour}.\mathsf{Flavour} \text{ iff } \mathsf{Beer}^{\mathcal{I}} \subseteq \{x \mid \exists y : \langle x, y\rangle \in \mathsf{hasFlavour}^{\mathcal{I}} \text{ and } y \in \mathsf{Flavour}^{\mathcal{I}}\}$$

Since (2.33) (the PropertyRange axiom of hasFlavour) already restricts that all the individuals which hasFlavour relates to must be from Flavour, (2.35) can be simplified as (2.36):

$$\mathsf{Beer} \sqsubseteq \exists \mathsf{hasFlavour} \tag{2.36}$$

Note that in (2.35) or (2.36), we only set Beer as a subclass of ∃hasFlavour.Flavour or ∃hasFlavour rather than making them equivalent (i.e. Beer ≡ ∃hasFlavour.Flavour). The reason is that by satisfying the equivalence, any individual which is related by hasFlavour to a Flavour is a Beer, which is too restricted as other foods (not a beer) might also have their flavours recorded.

Finally, a HasValue expression $\exists P.\{a\}$ denotes a set of individuals x which are related to a particular individual a by P, and it is interpreted as:

$$(\exists P.\{a\})^{\mathcal{I}} = \{x \mid \exists y : \langle x, y\rangle \in P^{\mathcal{I}} \text{ and } y = a^{\mathcal{I}}\}$$

For example, in the beer ontology, we may express every Porter has a Dark colour by:

$$\mathsf{Porter} \sqsubseteq \exists \mathsf{hasColour}.\{\mathsf{Dark}\} \tag{2.37}$$

Again, it would be inappropriate to specify Porter ≡ ∃hasColour.{Dark}, since individuals other than porters may also have a dark colour. However, we might adopt the use of IntersectionOf to give a more proper definition of Porter as things which are beers and have a dark colour by:

$$\mathsf{Porter} \equiv \mathsf{Beer} \sqcap \exists \mathsf{hasColour}.\{\mathsf{Dark}\} \tag{2.38}$$

Note that a HasValue expression $\exists P.\{a\}$ can be treated as a special SomeValuesFrom expression $\exists P.C_E$, where C_E is a class containing just one individual a. Note that from the viewpoint of inference, there is a significant difference between $C \sqsubseteq \exists P.\{a\}$ and $C \sqsubseteq \exists P.C_E$ (where C_E contains more than one individual). For $C \sqsubseteq \exists P.\{a\}$, every individual x in C derives an instance $\langle x, a\rangle$ to P. However, in the case of $C \sqsubseteq \exists P.C_E$, for each individual x in C, we only know it must be related to at lease one individual in C_E by P, but which one of them is unknown. Therefore, including axioms like $C \sqsubseteq \exists P.C_E$ in an ontology might result in the inference of anonymous individuals (i.e. non-deterministic reasoning).

Finally, specifying a class by enumerating its included individuals (i.e. $\{a_1 \ldots a_n\}$ in DL) is

achieved by the constructor OneOf, and this class is called an **anonymous class**, which is interpreted as:

$$(\{a_1 \dots a_n\})^{\mathcal{I}} = \{{a_1}^{\mathcal{I}}, \dots, {a_n}^{\mathcal{I}}\}$$

MinCardinality, MaxCardinality **and** ExactCardinality**:**

OWL 2 DL provides three cardinality-related constructors to express restrictions over the number of different objects to which subjects are related by a property. Expressions produced by constructors MinCardinality, MaxCardinality and ExactCardinality are denoted as unqualified as $\geqslant n\, P$, $\leqslant n\, P$ and $=n\, P$ (where n is a natural number), respectively, and $\mathcal{I}$ interprets them as:

$$(\geqslant n\, P)^{\mathcal{I}} = \{x \mid \#\{y \mid \langle x, y\rangle \in P^{\mathcal{I}}\} \geq n\}$$

$$(\leqslant n\, P)^{\mathcal{I}} = \{x \mid \#\{y \mid \langle x, y\rangle \in P^{\mathcal{I}}\} \leq n\}$$

$$(=n\, P)^{\mathcal{I}} = \{x \mid \#\{y \mid \langle x, y\rangle \in P^{\mathcal{I}}\} = n\}$$

As can be seen, $\geqslant n\, P$ denotes a set of x which is related to at least n different y by P. Similarly, $\leqslant n\, P$ specifies a set of x such that there are at most n different y to which P relates x. Finally, $=n\, P$ represents a set of x such that there are exactly n different y to which x is related via P. Cardinality-related expressions can also be qualified, expressed as $\geqslant n\, P.C_E$, $\leqslant n\, P.C_E$ and $=n\, P.C_E$ in DL. We provide their interpretations as:

$$(\geqslant n\, P.C_E)^{\mathcal{I}} = \{x \mid \#\{y \mid \langle x, y\rangle \in P^{\mathcal{I}} \text{ and } y \in {C_E}^{\mathcal{I}}\} \geq n\}$$

$$(\leqslant n\, P.C_E)^{\mathcal{I}} = \{x \mid \#\{y \mid \langle x, y\rangle \in P^{\mathcal{I}} \text{ and } y \in {C_E}^{\mathcal{I}}\} \leq n\}$$

$$(=n\, P.C_E)^{\mathcal{I}} = \{x \mid \#\{y \mid \langle x, y\rangle \in P^{\mathcal{I}} \text{ and } y \in {C_E}^{\mathcal{I}}\} = n\}$$

Interestingly, an ExactCardinality expression can be semantically represented as the intersection of a MinCardinality and a MaxCardinality expression (i.e. $(=n\, P)^{\mathcal{I}} = (\geqslant n\, P \sqcap \leqslant n\, P)^{\mathcal{I}}$ or $(=n\, P.C_E)^{\mathcal{I}} = (\geqslant n\, P.C_E \sqcap \leqslant n\, P.C_E)^{\mathcal{I}}$). Intuitively, $\exists P.C_E$ is also semantically equivalent to $\geqslant 1\, P.C_E$, and therefore, (2.35) can be alternatively written as:

$$\text{Beer} \sqsubseteq \geqslant 1\, \text{hasFlavour}.\text{Flavour} \tag{2.39}$$

2.3.3 Property-related Semantics

In this section, we continue with the semantics related to properties. Unlike class-related semantics, which are purely constructed between class expressions, OWL 2 standard groups some semantics, such as PropertyDomain, PropertyRange and ReflexiveProperty into property-related semantics, even if they are formed between class expressions. Property expressions can be **atomic** or **complex** (denoted as P or P_E). Note that P_E can be either P or P^-, as OWL 2 standard only provides one constructor InverseOf to form a complex property expression.

InverseOf and InverseProperty:

An InverseOf expression P^- swaps subjects and objects of P, and it is interpreted as:

$$(P^-)^{\mathcal{I}} = \{\langle y, x\rangle \mid \langle x, y\rangle \in P^{\mathcal{I}}\}$$

In OWL 2, a property can be defined as the inverse of another one, i.e. $P \equiv Q^-$, where we call P an InverseProperty of Q and vice versa. To satisfy $P \equiv Q^-$ is defined as:

$$\mathcal{I} \models P \equiv Q^- \text{ iff } P^{\mathcal{I}} = \{\langle y, x\rangle \mid \langle x, y\rangle \in Q^{\mathcal{I}}\}$$

In the beer ontology, we might add a property isFermentedBy, which relates beers to their fermentation yeasts, and another one fermentsBeer, which relates yeasts to their fermenting beers. We can express that the two properties are inverse of each other by:

$$\text{isFermentedBy} \equiv \text{fermentsBeer}^- \tag{2.40}$$

To satisfy this axiom, the following truth condition must hold:

$$\mathcal{I} \models \text{isFermentedBy} \equiv \text{fermentsBeer}^- \text{ iff } \text{isFermentedBy}^{\mathcal{I}} = \{\langle y, x\rangle \mid \langle x, y\rangle \in \text{fermentsBeer}^{\mathcal{I}}\}$$

PropertyDomain and PropertyRange:

In the previous section, we have shown examples of PropertyRange axioms, such as (2.33) and (2.34). Similar to PropertyRange axioms, which restrict the objects of a property to individuals from some particular class, PropertyDomain axioms apply a similar constraint over the subjects

of a property. To satisfy PropertyDomain and PropertyRange axioms are respectively specified as:

$$\mathcal{I} \models \top \sqsubseteq \forall P^{-}.C_E \text{ iff } \forall x : \text{if } \langle x, y\rangle \in P^{\mathcal{I}} \text{ then } x \in {C_E}^{\mathcal{I}}$$

$$\mathcal{I} \models \top \sqsubseteq \forall P.C_E \text{ iff } \forall y : \text{if } \langle x, y\rangle \in P^{\mathcal{I}} \text{ then } y \in {C_E}^{\mathcal{I}}$$

As can be seen, PropertyDomain and PropertyRange axioms are actually subsumption relations between class expressions (i.e. from $\top$ to $\forall P^{-}.C_E$ and from $\top$ to $\forall P.C_E$, respectively). However, OWL 2 standard considers them as property-related semantics rather than class-related. In the beer ontology, if we include all yeasts in a new class Yeast, we can specify the PropertyDomain and PropertyRange axioms for isFermentedBy and fermentsBeer by (2.41) – (2.44):

$$\top \sqsubseteq \forall \mathsf{isFermentedBy}^{-}.\mathsf{Beer} \qquad (2.41)$$

$$\top \sqsubseteq \forall \mathsf{isFermentedBy}.\mathsf{Yeast} \qquad (2.42)$$

$$\top \sqsubseteq \forall \mathsf{fermentsBeer}^{-}.\mathsf{Yeast} \qquad (2.43)$$

$$\top \sqsubseteq \forall \mathsf{fermentsBeer}.\mathsf{Beer} \qquad (2.44)$$

Note that specifying the PropertyDomain of P is equivalent to specifying the PropertyRange of P^{-}. The DL syntax for defining the PropertyDomain of P as C_E can be alternatively expressed as $\mathsf{Dom}(P, C_E)$, and similarly, setting the PropertyRange of P as C_E can be written as $\mathsf{Rng}(P, C_E)$.

There might be multiple class expressions defining the domain or range of a property (i.e. $\mathsf{Dom}(P, C_{E_1}), \ldots, \mathsf{Dom}(P, C_{E_n})$ or $\mathsf{Rng}(P, C_{E_1}), \ldots, \mathsf{Rng}(P, C_{E_n})$). In this case, we should note that this actually restricts subjects or objects to the intersection of these class expressions (defined by their interpretations below), and not the union of them.

$$(\mathsf{Dom}(P, C_{E_1}), \ldots, \mathsf{Dom}(P, C_{E_n}))^{\mathcal{I}} = (\mathsf{Dom}(P, C_{E_1} \sqcap \ldots \sqcap C_{E_n}))^{\mathcal{I}}$$

$$(\mathsf{Rng}(P, C_{E_1}), \ldots, \mathsf{Rng}(P, C_{E_n}))^{\mathcal{I}} = (\mathsf{Rng}(P, C_{E_1} \sqcap \ldots \sqcap C_{E_n}))^{\mathcal{I}}$$

Thus, in the beer ontology, restricting the PropertyDomain of hasFlavour and hasColour to be instances from Ale or Lager should be expressed as:

$$\mathsf{Dom}(\mathsf{hasFlavour}, \mathsf{Ale} \sqcup \mathsf{Lager}) \qquad (2.45)$$

$$\mathsf{Dom}(\mathsf{hasColour}, \mathsf{Ale} \sqcup \mathsf{Lager}) \qquad (2.46)$$

rather than as:

$$\mathsf{Dom}(\mathsf{hasFlavour}, \mathsf{Ale}) \qquad \mathsf{Dom}(\mathsf{hasColour}, \mathsf{Ale})$$

$$\mathsf{Dom}(\mathsf{hasFlavour}, \mathsf{Lager}) \qquad \mathsf{Dom}(\mathsf{hasColour}, \mathsf{Lager})$$

which actually mean $\mathsf{Dom}(\mathsf{hasFlavour}, \mathsf{Ale} \sqcap \mathsf{Lager})$ and $\mathsf{Dom}(\mathsf{hasColour}, \mathsf{Ale} \sqcap \mathsf{Lager})$.

SubPropertyOf, EquivalentProperties, and DisjointProperties:

Analogous to SubClassOf, EquivalentClasses and DisjointClasses, OWL 2 DL provides constructors SubPropertyOf, EquivalentProperties, and DisjointProperties to respectively define relations of subsumption, equivalence and disjointness among property expressions. In DL, one property expression stated as a sub property of another is expressed as $P_{E_1} \sqsubseteq P_{E_2}$, two properties equivalent to each other is denoted as $P_{E_1} \equiv P_{E_2}$, and $\mathsf{DisPro}(P_{E_1}, \ldots, P_{E_n})$ is used for specifying that some property expressions are pairwise disjoint. Note that although the form of $C_{E_1} \sqcap C_{E_2} \sqsubseteq \bot$ is a proper DL statement to denote two disjoint classes, a similar expression like $P_{E_1} \sqcap P_{E_2} \sqsubseteq \bot$ for specifying two disjoint properties is disallowed [Rud11] (as $\bot$ does not denote an empty set of the Cartesian product of the knowledge universe). Instead, the syntax of $\mathsf{DisPro}(P_{E_1}, \ldots, P_{E_n})$ should be used for representing DisjointProperties. We list the truth conditions for satisfying the three types of relations as:

$$\mathcal{I} \models P_{E_1} \sqsubseteq P_{E_2} \text{ iff } {P_{E_1}}^{\mathcal{I}} \subseteq {P_{E_2}}^{\mathcal{I}}$$

$$\mathcal{I} \models P_{E_1} \equiv P_{E_2} \text{ iff } {P_{E_1}}^{\mathcal{I}} = {P_{E_2}}^{\mathcal{I}}$$

$$\mathcal{I} \models \mathsf{DisPro}(P_{E_1}, \ldots, P_{E_n}) \text{ iff } {P_{E_i}}^{\mathcal{I}} \cap {P_{E_j}}^{\mathcal{I}} = \emptyset \text{ where } 1 \leq i < j \leq n$$

We can illustrate these constructors by adding the following axioms to the beer ontology.

$$\mathsf{hasColour} \sqsubseteq \mathsf{hasDescription} \quad (2.47)$$

$$\mathsf{hasFlavour} \sqsubseteq \mathsf{hasDescription} \quad (2.48)$$

$$\mathsf{hasFlavour} \equiv \mathsf{hasTaste} \quad (2.49)$$

$$\mathsf{DisPro}(\mathsf{hasTaste}, \mathsf{hasColour}) \quad (2.50)$$

where (2.47) and (2.48) are two SubPropertyOf axioms, which set hasColour and hasFlavour as two sub properties of a new property hasDescription, (2.49) defines another new property hasTaste equivalent to hasFlavour, and finally (2.50) specifies that hasTaste and hasColour are disjoint with each other.

PropertyChain and TransitiveProperty:

A new feature from OWL 2 compared with OWL 1 is the ability to concatenate a chain of properties by using the constructor PropertyChain (symbolised as $\circ$). Concatenating two properties P_{E_1} and P_{E_2} denotes a set of $\langle x, z \rangle$ such that $\langle x, y \rangle$ and $\langle y, z \rangle$ are respectively in P_{E_1}

and P_{E_2}. For instance, we might add to the beer ontology: 1) a property brewedIn to relate beers to the places where the beers are brewed; 2) a property locatedIn which relates places to their upper-level places; 3) two A-Box facts asserted as (2.51) and (2.52):

$$\mathsf{brewedIn}(\mathsf{MunichDunkel}, \mathsf{Munich}) \quad (2.51) \qquad \mathsf{locatedIn}(\mathsf{Munich}, \mathsf{Germany}) \quad (2.52)$$

Thus, if we have $\mathsf{brewedIn} \circ \mathsf{locatedIn}$ concatenating brewedIn and locatedIn, the pair of individuals $\langle \mathsf{MunichDunkel}, \mathsf{Germany} \rangle$ will be included in this PropertyChain expression. We might additionally specify a subsumption relationship from $\mathsf{brewedIn} \circ \mathsf{locatedIn}$ to brewedIn as:

$$\mathsf{brewedIn} \circ \mathsf{locatedIn} \sqsubseteq \mathsf{brewedIn} \quad (2.53)$$

Hence $\langle \mathsf{MunichDunkel}, \mathsf{Germany} \rangle$ will be derived as an instance of brewedIn because of (2.53). The most general PropertyChain expression is to concatenate an arbitrary number of property expressions, which is denoted as $P_{E_1} \circ P_{E_2} \circ \ldots \circ P_{E_n}$, and $\mathcal{I}$ interprets this as:

$$(P_{E_1} \circ P_{E_2} \circ \ldots \circ P_{E_n})^{\mathcal{I}} = \{\langle x_0, x_n \rangle \mid \langle x_0, x_1 \rangle \in {P_{E_1}}^{\mathcal{I}}, \langle x_1, x_2 \rangle \in {P_{E_2}}^{\mathcal{I}}, \ldots, \langle x_{n-1}, x_n \rangle \in {P_{E_n}}^{\mathcal{I}}\}$$

On the other hand, a special usage of PropertyChain is to make an atomic property as a super property of a PropertyChain expression concatenating itself (i.e. $P \circ P \sqsubseteq P$). This gives the property the semantics of **transitivity**, which means that if $\langle x, y \rangle$ and $\langle y, z \rangle$ are in P, then $\langle x, z \rangle$ is also its instance, and P is called a TransitiveProperty. The truth condition for satisfying a transitive property P is:

$$\mathcal{I} \models P \circ P \sqsubseteq P \text{ iff } \forall x, y, z : \text{if} \langle x, y \rangle \in P \text{ and } \langle y, z \rangle \in P \text{ then } \langle x, z \rangle \in P$$

Note that specifying a transitive property can be alternatively written as $\mathsf{Tra}(P)$ in DL. Thus in the beer ontology, we can set locatedIn as a TransitiveProperty by (2.54) or (2.55):

$$\mathsf{locatedIn} \circ \mathsf{locatedIn} \sqsubseteq \mathsf{locatedIn} \quad (2.54) \qquad \mathsf{Tra}(\mathsf{locatedIn}) \quad (2.55)$$

Considering the A-Box fact (2.52) and another fact (2.56) below, we can derive a new fact (2.57) because of the transitivity of locatedIn.

$$\mathsf{locatedIn}(\mathsf{Germany}, \mathsf{Europe}) \quad (2.56) \qquad \mathsf{locatedIn}(\mathsf{Munich}, \mathsf{Europe}) \quad (2.57)$$

The transitivity can be also understood as a characteristic of a property P. Other allowed property characteristics are SymmetricProperty, FunctionalProperty, ReflexiveProperty, *etc.*, which we now define.

SymmetricProperty **and** AsymmetricProperty**:**

A property P is a SymmetricProperty when it is the case that if $\langle x, y\rangle$ is in P, then $\langle y, x\rangle$ also belongs to P. We denote a symmetric property by $P \equiv P^-$ or a simpler syntax $\mathsf{Sym}(P)$, and satisfying a SymmetricProperty P is defined as:

$$\mathcal{I} \models P \equiv P^- \text{ iff } \forall x, y : \text{if } \langle x, y\rangle \in P^{\mathcal{I}} \text{ then } \langle y, x\rangle \in P^{\mathcal{I}}$$

In the beer ontology, we could have a SymmetricProperty named adjacentPlace by (2.58) or (2.59) to relate places to their adjacent places. Intuitively, the symmetry of adjacentPlace expresses that if a place x is adjacent to a place y, then y must be adjacent to x.

$$\mathsf{adjacentPlace} \equiv \mathsf{adjacentPlace}^- \quad (2.58) \qquad \mathsf{Sym}(\mathsf{adjacentPlace}) \quad (2.59)$$

As Germany and Belgium are adjacent to each other, if we assert one of (2.60) and (2.61), the other will be automatically inferred as a result of the symmetry.

$$\mathsf{adjacentPlace}(\mathsf{Germany}, \mathsf{Belgium}) \quad (2.60) \qquad \mathsf{adjacentPlace}(\mathsf{Belgium}, \mathsf{Germany}) \quad (2.61)$$

An AsymmetricProperty by contrast means that if $\langle x, y\rangle$ is in a property P, then $\langle y, x\rangle$ must not be an instance of P; in other words, P must not contain $\langle x, y\rangle$ and $\langle y, x\rangle$ at the same time. In DL, the asymmetry of a property is specified by making the property disjoint with its InverseOf expression, denoted by $\mathsf{DisPro}(P, P^-)$ or $\mathsf{Asy}(P)$. The truth condition for satisfying an AsymmetricProperty P is defined as:

$$\mathcal{I} \models \mathsf{DisPro}(P, P^-) \text{ iff } \forall x, y : \text{if } \langle x, y\rangle \in P^{\mathcal{I}} \text{ then } \langle y, x\rangle \notin P^{\mathcal{I}}$$

In the beer ontology, since hasDescription relates beers to their descriptions, which cannot be inversely interpreted, this property should be specified asymmetric by (2.62) or (2.63).

$$\mathsf{DisPro}(\mathsf{hasDescription}, \mathsf{hasDescription}^-) \quad (2.62) \qquad \mathsf{Asy}(\mathsf{hasDescription}) \quad (2.63)$$

Note that from (2.62) or (2.63), hasFlavour (also its equivalent property hasTaste) and hasColour, which are sub properties of hasDescription, will implicitly obtain the semantics of asymmetry. Indeed, if any sub property of an AsymmetricProperty P contains symmetric tuples (i.e. $\langle x, y\rangle$ and $\langle y, x\rangle$), these tuples will be derived as instances of P, which conflicts with the asymmetry.

ReflexiveProperty **and** IrreflexiveProperty**:**

The semantics of **reflexivity** mean that an individual is related to itself by some property. OWL 2 provides a HasSelf expression $\exists P.\mathsf{Self}$, formed by using the special class Self, to denote the individuals relating to themselves by P. $\mathcal{I}$ interprets $\exists P.\mathsf{Self}$ as:

$$(\exists P.\mathsf{Self})^{\mathcal{I}} = \{x \mid \langle x, x\rangle \in P^{\mathcal{I}}\}$$

The reflexivity can be either universally or locally specified. Universal reflexivity expresses that all individuals in the knowledge domain (i.e. Δ) are related to themselves by a property. In DL, the universal reflexivity is specified by setting $\top$ as a subclass of $\exists P.\mathsf{Self}$, i.e. $\top \sqsubseteq \exists P.\mathsf{Self}$, or alternatively be written as $\mathsf{Ref}(P)$. We call such a property P a ReflexiveProperty, and satisfying a ReflexiveProperty P is defined as:

$$\mathcal{I} \models \top \sqsubseteq \exists P.\mathsf{Self} \text{ iff } \forall x \in \Delta : \langle x, x\rangle \in P$$

Local reflexivity (*a.k.a.* SelfRestriction) by contrast is expressed as $C \sqsubseteq \exists P.\mathsf{Self}$, which specifies that only individuals from a local class C (i.e. not all individuals from the knowledge universe) are related to themselves by P. Satisfying local reflexivity should follow:

$$\mathcal{I} \models C \sqsubseteq \exists P.\mathsf{Self} \text{ iff } \forall x \in C : \langle x, x\rangle \in P$$

We can even make C equivalent to a HasSelf expression (i.e. $C \equiv \exists P.\mathsf{Self}$); consequently, not only any individual x in C will infer a tuple $\langle x, x\rangle$ to P, but also any tuple of the form $\langle y, y\rangle$ of P will derive y as a member of C.

Irreflexivity is the negation of reflexivity, and an IrreflexiveProperty P, denoted as $\top \sqsubseteq \neg\exists P.\mathsf{Self}$ or $\mathsf{Irr}(P)$, must not relate any individual x to the same individual itself. $\mathcal{I}$ satisfies an IrreflexiveProperty P on the following condition:

$$\mathcal{I} \models \top \sqsubseteq \neg\exists P.\mathsf{Self} \text{ iff } \forall x \in \Delta : \langle x, x\rangle \notin P^{\mathcal{I}}$$

The property hasDescription in the beer ontology can be naturally specified with the semantics of irreflexivity by (2.64) or (2.65), because it is not semantically sensible to relate a beer to itself through hasDescription.

$$\top \sqsubseteq \neg\exists \mathsf{hasDescription}.\mathsf{Self} \qquad (2.64) \qquad\qquad \mathsf{Irr}(\mathsf{hasDescription}) \qquad (2.65)$$

Analogous to asymmetry, the irreflexivity of hasDescription is inherited by its sub properties hasFlavour, hasTaste and hasColour.

FunctionalProperty and InverseFunctionalProperty:

FunctionalProperty is a special case of using the MaxCardinality, where each subject in a functional property is related to at most one object through this property. Denoting P as a FunctionalProperty uses the DL syntax $\top \sqsubseteq \leqslant 1\, P$ or Fun(P). A functional property should satisfy the following condition:

$$\mathcal{I} \models \top \sqsubseteq \leqslant 1\, P \text{ iff } \forall x, y, z : \text{if } \langle x, y\rangle \in P^{\mathcal{I}} \text{ and } \langle x, z\rangle \in P^{\mathcal{I}} \text{ then } y = z$$

To explain the condition, it is necessary to introduce another constructor SameIndividual, which can be used to specify that multiple individuals refer to the same knowledge object, even though their names are different. Recall that in Definition 2.1, the interpretation $\mathcal{I}$ associates each individual a with an object a in the knowledge domain. Expressing multiple individuals are the same, expressed as $=(a_1, \ldots, a_n)$, means that $\mathcal{I}$ maps these individuals to the same object:

$$\mathcal{I} \models =(a_1, \ldots, a_n) \text{ iff } {a_i}^{\mathcal{I}} = {a_j}^{\mathcal{I}} \text{ where } 1 \leq i < j \leq n$$

Note that this contradicts the UNA, which requires that individuals with different names do not refer to the same object. Specifying that individuals do not refer to the same object can use DifferentIndividuals, denoted as $\neq(a_1, \ldots, a_n)$, for which the satisfying condition is:

$$\mathcal{I} \models \neq(a_1, \ldots, a_n) \text{ iff } {a_i}^{\mathcal{I}} \neq {a_j}^{\mathcal{I}} \text{ where } 1 \leq i < j \leq n$$

by which $\mathcal{I}$ maps individuals $a_1, \ldots, a_n$ to all different objects in the knowledge domain. Applying DifferentIndividuals over all individuals in an ontology is semantically equivalent to applying the UNA. Considering the truth condition for satisfying $\top \sqsubseteq \leqslant 1\, P$, it defines that if the same individual is related to two individuals by P, then the two individuals are the same (i.e. they are assigned to the same object in the knowledge universe).

If the inverse of P is a FunctionalProperty, we call P an InverseFunctionalProperty, denoted as $\top \sqsubseteq \leqslant 1\, P^-$ or InvFun(P) and satisfying this should follow the following condition:

$$\mathcal{I} \models \top \sqsubseteq \leqslant 1\, P^- \text{ iff } \forall x, y, z : \text{if } \langle y, x\rangle \in P^{\mathcal{I}} \text{ and } \langle z, x\rangle \in P^{\mathcal{I}} \text{ then } y = z$$

HasKey:

The constructor HasKey is provided by OWL 2 DL, but not in the $\mathcal{SROIQ}$ DL, and it allows multiple properties $P_1 \dots P_n$ together to uniquely identify an individual in a class C, which is denoted as HasKey$(C, (P_1 \dots P_n))$. The satisfying condition is:

$$\mathcal{I} \models \text{HasKey}(C, (P_1 \dots P_n)) \text{ iff } \forall x, y, z_1, \dots, z_n :$$
$$\text{if } \langle x, z_1 \rangle \in {P_1}^{\mathcal{I}}, \dots, \langle x, z_n \rangle \in {P_n}^{\mathcal{I}} \text{and } \langle y, z_1 \rangle \in {P_1}^{\mathcal{I}}, \dots, \langle y, z_n \rangle \in {P_n}^{\mathcal{I}} \text{ then } x = y$$

which means that if both x and y are related to $z_1 \dots z_n$ by $P_1 \dots P_n$, respectively, then x and y are the same.

NegativePropertyAssertion:

Finally, OWL 2 DL allows users to explicitly specify that a property P does not contain $\langle x, y \rangle$ as its instance (i.e. $\neg P(x, y)$); therefore $\mathcal{I}$ does not assign such a tuple to this property:

$$\mathcal{I} \models \neg P(x, y) \text{ iff } \langle x, y \rangle \notin P^{\mathcal{I}}$$

2.4 Summary

In this chapter, we have reviewed the complete Direct Semantics of the most recent release of the OWL 2 language by using the interpretation $\mathcal{I}$ of this semantics. We choose the Direct Semantics rather than the RDF-based Semantics to ensure the inference decidability. One reason why the RDF-based Semantics brings undecidable reasoning is because it interprets classes as members of the knowledge universe, while the Direct Semantics interprets classes as subsets of the knowledge universe. Therefore, the RDF-based semantics allows the case of specifying a class as a member of another class, which is disallowed in the Direct Semantics. The Direct Semantics plays a significant role in ontology inference, the details of which are reviewed in the next chapter.

Chapter 3

Review of Inference

3.1 Introduction

In Section 2.2.3 of Chapter 2, we have provided a brief overview of **inference** (*a.k.a.* **reasoning**), which is the process of deriving implicit consequences from explicitly expressed knowledge. Being able to perform inference is usually considered as a key difference of OWL from more traditional database modelling languages.

Inference over ontologies can be divided into T-Box and A-Box inference, where the **T-Box inference** focuses on the schema of the ontology, and the **A-Box inference** concentrates on the data-related inference. In each category, different **inference tasks** are considered; for instance, *classification of atomic classes*, which builds the hierarchy of all classes in an ontology, belongs to the T-Box inference, and *realisation*, which derives for each individual its most specific membership of classes, is a task of the A-Box inference. A system that is able to perform some of (or all) the inference tasks is called a **reasoner**. To perform inference, different inference techniques, such as the **tableaux algorithm** and **rule-based inference**, are adopted by reasoners.

In this chapter, Section 3.2 first introduces inference tasks that are commonly considered in OWL 2 DL. Then, Section 3.3 reviews the tableaux algorithm, and analyses its advantage of complete T-Box inference, and disadvantages of intractable A-Box inference. Next, Section 3.4 reviews rule-based inference which aims for tractable inference over large A-Boxes.

3.2 Inference Tasks

For a given ontology O and an *axiom*, we denote by $O \models axiom$ that this *axiom* can be inferred from existing statements in O based on some semantics. We use $O \not\models axiom$ for the case that the *axiom* cannot be inferred from the existing statements contained in O based on those semantics. Since we adopt the OWL 2 interpreted by Direct Semantics (i.e. OWL 2 DL), the inference we consider is based on the interpretations of OWL 2 expressions and truth conditions for holding OWL 2 axioms defined by the interpretation $\mathcal{I}$ in Definition 2.1. Moreover, an inference task is **global**, if it considers the whole ontology as input. In OWL 2 DL, global inference tasks include *global consistency*, *classification of atomic classes* and *realisation*. By contrast, **local** inference tasks take a fragment of an ontology as input, and they include *subclass relationship*, *class equivalence*, *class satisfiability*, *instance checking* and *instance retrieval*.

3.2.1 Global Inference Tasks

Global consistency

An ontology O is said to be **consistent**, if it has at least one interpretation which satisfies all axioms contained in this ontology; such an interpretation is also called a **model** of O. Because we adopt the Direct Semantics[1] of OWL 2 (i.e. the $\mathcal{I}$ of OWL 2 DL which we have reviewed in Chapter 2), *global consistency* can be understood as checking as to whether all axioms in O can be satisfied based on $\mathcal{I}$. An **inconsistent** ontology contains at least one axiom that cannot be satisfied by $\mathcal{I}$; for example, from (2.64), $\mathcal{I}$ satisfies that hasDescription is an IrreflexiveProperty if hasDescription does not relate any individuals to themselves. Therefore, adding any fact of the form hasDescription(x, x) will result the beer ontology entering into an inconsistent state. As *global consistency* is a global inference task, it requires considering T-Box axioms and A-Box facts together.

Classification of atomic classes

[1] We adopt $\mathcal{I}$ of OWL 2 DL as the ontology model in this thesis.

Atomic classes, such as PaleAle, Porter and Lager, are atomic concepts which denote sets of individuals. They are not like complex class expressions (e.g. $C_{E_1} \sqcap C_{E_2}$, $C_{E_1} \sqcup C_{E_2}$ and $\forall P.C_E$), which are formed by combining atomic classes and constructors. The classification of them is the process of computing the hierarchy of all atomic classes included in an ontology; in other words, the derivation of all subsumption relations among all the atomic classes. Classification of atomic class is a T-Box inference task, as it focuses on the schema of an ontology.

Realisation

While the previous task classifies all atomic classes, **realisation** can be described as the process of classifying all individuals. This task derives for every individual its most specific atomic classes. An atomic class C is more specific than another one D, if C is the subclass of D (i.e. $C \sqsubseteq D$). In order to demonstrate this inference task, we add to the beer ontology the following T-Box axioms:

$$\mathsf{BrownBritishBeer} \sqsubseteq \mathsf{Ale} \tag{3.1}$$

$$\mathsf{BrownBritishBeer} \equiv \mathsf{Beer} \sqcap \exists \mathsf{hasColour}.\{\mathsf{Brown}\} \sqcap \exists \mathsf{brewedIn}.\{\mathsf{Britain}\} \tag{3.2}$$

which introduce a class BrownBritishBeer as a subclass of Ale, and define this class as British-brewed beers that have a Brown colour.

For the beer individual BritishBrownAle, since we have already asserted it as an Ale by (2.3), it should be implicitly derived as a Beer, because every ale is a beer defined by (2.24). If we have another two A-Box facts (3.3) and (3.4):

$$\mathsf{hasColour}(\mathsf{BritishBrownAle}, \mathsf{Brown}) \tag{3.3}$$

$$\mathsf{brewedIn}(\mathsf{BritishBrownAle}, \mathsf{Britain}) \tag{3.4}$$

we can derive BritishBrownAle as a BrownBritishBeer (i.e. $O \models \mathsf{BrownBritishBeer}(\mathsf{BritishBrownAle})$) because of (3.2). Thus, through the task of realisation, the membership of BritishBrownAle is now more specific to BrownBritishBeer than to Ale. Note that, this example only shows the *realisation* over one individual; however, as a global inference task, *realisation* should repeat for every individual in an ontology.

3.2.2 Local Inference Tasks

Subclass relationship and *Class equivalence*

The inference task **subclass relationship** checks for a given ontology O and a SubClassOf axiom $C \sqsubseteq D$ (between two atomic classes) whether $O \models C \sqsubseteq D$ or $O \not\models C \sqsubseteq D$. For example, in the beer ontology, the two SubClassOf axioms (3.5) and (3.6) below are satisfied (i.e. $O \models \mathsf{Ale} \sqsubseteq \mathsf{LiquidBread}$ and $O \models \mathsf{Lager} \sqsubseteq \mathsf{LiquidBread}$) because Ale and Lager are both subclasses of Beer defined by (2.24) and (2.25), and Beer and LiquidBread are specified equivalent by (2.28).

$$\mathsf{Ale} \sqsubseteq \mathsf{LiquidBread} \tag{3.5}$$

$$\mathsf{Lager} \sqsubseteq \mathsf{LiquidBread} \tag{3.6}$$

Besides Ale and Porter, all subclasses of Beer (e.g. PaleAle and BrownBritishBeer) are also subclasses of LiquidBread (i.e. equivalent classes share the same subclasses). The subsumption $\mathsf{PaleAle} \sqsubseteq \mathsf{Lager}$ is an example of $O \not\models C \sqsubseteq D$ (i.e. $O \not\models \mathsf{PaleAle} \sqsubseteq \mathsf{Lager}$); PaleAle as a subclass of Ale cannot be a subclass of Lager because Ale is disjoint with Lager defined by (2.29).

The inference task **class equivalence** determines for a given ontology O and a class equivalence axiom $C \equiv D$ (between two atomic classes) whether $O \models C \equiv D$ or $O \not\models C \equiv D$. Since $C \equiv D$ can be interpreted as $C \sqsubseteq D$ and $D \sqsubseteq C$, checking a *class equivalence* can be transformed as checking its equivalent two *subclass relationships*. Because both *subclass relationship* and *class equivalence* focus the structure of an ontology, we classify the two tasks into T-Box inference.

Class satisfiability

A class C is **satisfiable** if it is not empty (i.e. $C \not\equiv \bot$); and a class which cannot contain any individual is called **unsatisfiable** (i.e. $C \sqsubseteq \bot$). For example, if we make a class SpecialBeer a subclass of both Ale and Lager (i.e. $\mathsf{SpecialBeer} \sqsubseteq \mathsf{Ale}$, $\mathsf{SpecialBeer} \sqsubseteq \mathsf{Lager}$), from (2.29), we will infer $\mathsf{SpecialBeer} \sqsubseteq \bot$, and therefore, SpecialBeer is unsatisfiable. Thus this inference task checks whether a given class (except Nothing) is satisfiable or unsatisfiable.

Instance checking and *Instance retrieval*

As we have already illustrated in Section 2.2.3 of Chapter 2, these two inference tasks are

considered in A-Box inference. **Instance checking** is the process of determining for a given class C and a given individual a, whether a belongs to C or not (i.e. to check $O \models C(a)$ or $O \not\models C(a)$). The checking could result in unknown as OWL 2 follows the OWA, so true means $O \models C(a)$, unknown and false means $O \not\models C(a)$. In some circumstances, such as computing answers to a query executed over an ontology, it is more useful and efficient to retrieve all instances of a class or property, rather than checking whether a particular instance belongs to a class or property. The task of **instance retrieval** fits this case, and it derives for each class all of its individuals. Instance retrieval is quite useful in many practical applications, and to return all the instances belonging to a particular class is a frequent computation task when querying the ontological data.

3.2.3 Inference Properties of Reasoners

Software which is able to perform all or some of the above inference tasks is called a **reasoner**. Reasoners based on different inference techniques can be divided into different groups, such as tableaux-based reasoners and rule-based reasoners, which are detailed in Section 3.3 and Section 3.4, respectively. For a set of inference tasks performed by a reasoner, the three inference properties listed below are often used for evaluating the reasoner.

- **Soundness**:

  A reasoner $\mathcal{R}$ is **sound** for a given ontology O and a set of inference tasks $\mathcal{T}$, if O entails all the results computed by $\mathcal{R}$ for $\mathcal{T}$. We may denote this by $O \models \{axioms\}_{\mathcal{RT}}$, where $\{axioms\}_{\mathcal{RT}}$ is the set of derivations computed by $\mathcal{R}$ w.r.t. O and $\mathcal{T}$.

- **Completeness**:

  A reasoner $\mathcal{R}$ is **complete** for a given ontology O and a set of inference tasks $\mathcal{T}$, if $\mathcal{R}$ is able to derive all possible consequences for the tasks $\mathcal{T}$ entailed by O. We denote this by $O'_{\mathcal{T}} \subseteq \{axioms\}_{\mathcal{RT}}$, where $O'_{\mathcal{T}}$ is the set of consequences for $\mathcal{T}$ entailed by O, and $\{axioms\}_{\mathcal{RT}}$ is the set of results computed by $\mathcal{R}$ w.r.t. O and $\mathcal{T}$.

- **Complexity**:

For a given ontology O, the complexity for computing a set of inference tasks $\mathcal{T}$ by a reasoner $\mathcal{R}$ is also an important property for evaluating $\mathcal{R}$. We focus on **tractable** inference, which can be described as $\mathcal{R}$ is able to process $\mathcal{T}$ for O in polynomial time. Tractable inference is particularly important in practical inference systems. Since the OWL 2 DL is decidable, but can be N2EXPTIME-complete, three profiles (i.e. sub-languages) of OWL 2 (i.e. OWL 2 EL, OWL 2 QL and OWL 2 RL) are introduced to limit the full use of OWL 2, so that tractable inference can be achieved. We review the three profiles in Section 3.4.1.

3.3 Tableaux-based Inference

Tableaux-based inference relies on the tableaux algorithm, which reduces every inference task to a problem of consistency checking. The algorithm verifies the consistency by attempting to construct a tableau which represents a model for the ontology; the ontology is consistent if the constructing process terminates without finding any inconsistency. The tableaux algorithm is known to provide sound and complete inference for ontologies expressed in the $\mathcal{SROIQ}$ DL, on which the OWL 2 DL is based. Illustrating how the tableaux algorithm can be used to reason the full $\mathcal{SROIQ}$ DL is very complicated, so instead we choose the $\mathcal{ALC}$ DL, which is a simple fragment of $\mathcal{SROIQ}$ DL that only allows class expressions constructed by $\top$, $\bot$, $\neg$, $\sqcap$, $\sqcup$, $\forall$ and $\exists$. In this section, we first outline the $\mathcal{ALC}$ DL, and afterwards use it to demonstrate the tableaux algorithm.

3.3.1 $\mathcal{ALC}$ ontologies

$\mathcal{ALC}$ defines a subset of ontologies expressed in $\mathcal{SROIQ}$ DL, which is the logical underpinning of the Direct Semantics. Each of the DL acronyms (e.g. $\mathcal{A}$, $\mathcal{C}$ and $\mathcal{Q}$) corresponds to some DL constructors and axiom types, and a full list of the DL acronyms is summarised in Table 3.1. As can be seen, $\mathcal{ALC}$ ontologies allow the use of atomic classes (including Thing and Nothing), complex classes (using IntersectionOf, UnionOf, ComplementOf, AllValuesFrom and

Table 3.1: The family of DL acronyms

code	*corresponding DL constructors and axiom types*
$\mathcal{AL}$	- atomic class, top class $\top$ - complex classes $C \sqcap D$, $\forall P.C$, $\exists P$ - subclass ($C \sqsubseteq D$) and class equivalence ($C \equiv D$) axioms - assertions of the form $C(a)$, $P(a,b)$, $=(a,b)$, and $\neq(a,b)$
$\mathcal{ALC}$	$\mathcal{AL}$ plus: - bottom class $\bot$ - complex classes C, $(C \sqcup D)$, $\exists P.C$ - class disjunction $\mathsf{DisCla}(C_1, \ldots, C_n)$, and class disjoint union $\mathsf{DisUni}(D, C_1, \ldots, C_n)$
$\mathcal{S}$	$\mathcal{ALC}$ plus: - property transitivity $\mathsf{Tra}(P)$
$\mathcal{H}$	- subproperty ($P \sqsubseteq Q$) and property equivalence ($P \equiv Q$) axioms
$\mathcal{R}$	$\mathcal{H}$ plus: - top property and bottom property - local reflexivity restriction $\exists P.\mathsf{Self}$ - property disjunction $\mathsf{DisPro}(P_1, \ldots, P_n)$ - reflexivity $\mathsf{Ref}(P)$, irreflexivity $\mathsf{Irr}(P)$ of properties - symmetry $\mathsf{Sym}(P)$, and asymmetry $\mathsf{Asy}(P)$ of properties - chain subproperty axioms $P_1 \circ \cdots \circ P_n \sqsubseteq Q$ - negative property assertions: $\neg P(a,b)$
$\mathcal{O}$	- has-value restrictions $\exists P.\{a\}$, nominals $\{a\}$ (i.e. OneOf)
$\mathcal{I}$	- inverse properties P^-
$\mathcal{F}$	- functional restriction $\leqslant 1\, P$
$\mathcal{N}$	- unqualified cardinality restrictions $\leqslant n\, P$, $=n\, P$, $\geqslant n\, P$
$\mathcal{Q}$	- qualified cardinality restrictions $\leqslant n\, P.C$, $=n\, P.C$, $\geqslant n\, P.C$

SomeValuesFrom), axioms (expressing SubClassOf, EquivalentClasses and DisjointClasses), and assertions (specifying ClassAssertion, PropertyAssertion, SameIndividual and DifferentIndividuals).

3.3.2 Tableaux algorithm

In principle, the tableaux algorithm reasons an ontology by attempting to establish a tableau for the ontology, and such a tableau exists if the establishment process terminates without deriving any contradiction (i.e. inconsistency). As a result, inference becomes the task of checking whether an ontology is consistent or not, and indeed the inference tasks listed in Section 3.2 can all be reduced to a task of consistency checking. Before giving the detailed description of the tableaux algorithm, we first outline how inference tasks can be reduced. Moreover, to simplify the illustration of the tableaux algorithm, we adopt the **Negation Normal Form** (**NNF**) [Hor97] of an $\mathcal{ALC}$ ontology, which is logically equivalent to the ontology.

Reducing Inference Tasks

The tableaux algorithm uses the fact that inference tasks in Section 3.2 can be reduced to checking whether an ontology is consistent or not. For example, for the inference task of *Subclass relationship*, verifying $O \models C \sqsubseteq D$ is logically equivalent to verifying that $O \cup \{C(x) \sqcap \neg D(x)\}$ (where x is a new individual that does not appear in O) is inconsistent. Intuitively, for the inference task of *Class equivalence*, $O \models C \equiv D$ holds if and only if both $O \models C \sqsubseteq D$ and $O \models D \sqsubseteq C$ hold, which can be eventually reduced as two consistency checking problems. We summarise below how other local inference tasks can be translated to the task of consistency checking (the three global inference tasks can be reduced to repeated local inference tasks; for example, the task of *Classification of atomic classes*, which finds all subsumption relations between atomic classes, can be processed by repeatedly performing the *Subclass relationship* for every possible subsumption relation in the ontology).

- *Class consistency*: $O \models C \sqsubseteq \bot$ if and only if $O \cup \{C(x)\}$ is inconsistent, where x is a new individual that does not appear in O.
- *Instance checking*: $O \models C(a)$ if and only if $O \cup \{\neg C(a)\}$ is inconsistent.
- *Instance retrieval*: retrieving all instances of a class C needs to check for all individuals a_i in O, whether $O \models C(a_i)$, each of which is an instance checking task.

Negation Normal Form

In order to simplify the demonstration of the tableaux algorithm, we list the NNF of the $\mathcal{ALC}$ ontologies in Table 3.2. An ontology O is logically equivalent to its NNF O_{NNF}. For example, a subsumption $C_{E_1} \sqsubseteq C_{E_2}$ and its $(\neg C_{E_1} \sqcup C_{E_2})_{NNF}$ are respectively interpreted as $(\forall x)(C_{E_1}(x) \rightarrow C_{E_2}(x))$ and $(\forall x)(\neg C_{E_1}(x) \vee C_{E_2}(x))$, which are logically equivalent.

Tableaux algorithms for $\mathcal{ALC}$

The tableau of an $\mathcal{ALC}$ ontology is defined in Definition 3.1.

Table 3.2: $\mathcal{ALC}$ to NNF

$\mathcal{ALC}$	$\overset{NNF}{\Longrightarrow}$	$\mathcal{ALC}_{NNF}$
$C_{E_1} \sqsubseteq C_{E_2}$	$\overset{NNF}{\Longrightarrow}$	$(\neg C_{E_1} \sqcup C_{E_2})_{NNF}$
C	$\overset{NNF}{\Longrightarrow}$	C
$\neg C$	$\overset{NNF}{\Longrightarrow}$	$\neg C$
$\neg\neg C_E$	$\overset{NNF}{\Longrightarrow}$	$(C_E)_{NNF}$
$C_{E_1} \sqcup C_{E_2}$	$\overset{NNF}{\Longrightarrow}$	$(C_{E_1})_{NNF} \sqcup (C_{E_2})_{NNF}$
$C_{E_1} \sqcap C_{E_2}$	$\overset{NNF}{\Longrightarrow}$	$(C_{E_1})_{NNF} \sqcap (C_{E_2})_{NNF}$
$\neg(C_{E_1} \sqcup C_{E_2})$	$\overset{NNF}{\Longrightarrow}$	$(\neg C_{E_1})_{NNF} \sqcap (\neg C_{E_2})_{NNF}$
$\neg(C_{E_1} \sqcap C_{E_2})$	$\overset{NNF}{\Longrightarrow}$	$(\neg C_{E_1})_{NNF} \sqcup (\neg C_{E_2})_{NNF}$
$\forall P_E.C_E$	$\overset{NNF}{\Longrightarrow}$	$\forall P_E.(C_E)_{NNF}$
$\exists P_E.C_E$	$\overset{NNF}{\Longrightarrow}$	$\exists P_E.(C_E)_{NNF}$
$\neg\forall P_E.C_E$	$\overset{NNF}{\Longrightarrow}$	$\exists P_E.(\neg C_E)_{NNF}$
$\neg\exists P_E.C_E$	$\overset{NNF}{\Longrightarrow}$	$\forall P_E.(\neg C_E)_{NNF}$

Definition 3.1 *A tableau of an $\mathcal{ALC}$ ontology consists of:*

- *a set of nodes, each of which represents an individual (or an anonymous individual)*
- *a set of directed edges between pairs of nodes: if individuals represented by the nodes are related by a property, and the corresponding edge represents the property*
- *for each node x, a set $L(x)$ of class expressions, all containing x as their class instance*
- *for each pair of nodes $\langle x, y\rangle$, a set $L(x, y)$ of property expressions, all containing $\langle x, y\rangle$ as their property instance*

Suppose we have an $\mathcal{ALC}$ ontology $O_{\mathcal{ALC}} = \{C(a), C \sqsubseteq \exists P.D, D \sqsubseteq E\}$, and its NNF is $\{C(a), \neg C \sqcup \exists P.D, \neg D \sqcup E\}$. Since now there is only one present individual a, and the class expression including a is just C, the tableau of the $O_{\mathcal{ALC}}$ is initialised as:

$$\circ\, a \qquad L(a) = \{C\}$$

where there is one node a representing the individual a, and the $L(a)$ now only includes C. If we consider an instance checking that whether $O_{\mathcal{ALC}} \models (\exists P.E)(a)$ or $O_{\mathcal{ALC}} \not\models (\exists P.E)(a)$, the tableaux algorithm will first reduce this inference task to a consistency checking problem, i.e. to check $O_{\mathcal{ALC}} \cup \neg(\exists P.E)(a)$ is consistent or not; if not, then $O_{\mathcal{ALC}} \models (\exists P.E)(a)$ will be a logical consequence from $O_{\mathcal{ALC}}$. Since the NNF of $\neg(\exists P.E)(a)$ is $\forall P.(\neg E)(a)$, then $O_{\mathcal{ALC}} \cup \neg(\exists P.E)(a)$

can be denoted in NNF as $\{C(a), \forall P.(\neg E)(a), \neg C \sqcup \exists P.D, \neg D \sqcup E\}$. Consequently, the tableau of $O_{\mathcal{ALC}} \cup \neg(\exists P.E)(a)$ is updated to include $\forall P.(\neg E)$ in $L(a)$:

$$\circ\, a \qquad L(a) = \{C, \forall P.(\neg E)\}$$

Next, we consider $\neg C \sqcup \exists P.D$. Firstly, $\neg C \sqcup \exists P.D$ denotes a T-Box axiom, which should be applicable for every individual (including a), so $L(a)$ is updated to $\{C, \forall P.(\neg E), \neg C \sqcup \exists P.D\}$:

$$\circ\, a \qquad L(a) = \{C, \forall P.(\neg E), \neg C \sqcup \exists P.D\}$$

Secondly, because this expression denotes a semantics of disjunction (i.e. **non-determinism**), either $\neg C$ or $\exists P.D$ can be added to $L(a)$. However, the option of adding $\neg C$ into $L(a)$ immediately generates a contradiction, because C is already in $L(a)$ (i.e. $\neg C$ contradicts C). Therefore, we choose the other option which adds $\exists P.D$ into $L(a)$, and the $L(a)$ becomes $\{C, \forall P.(\neg E), \neg C \sqcup \exists P.D, \exists P.D\}$:

$$\circ\, a \qquad L(a) = \{C, \forall P.(\neg E), \neg C \sqcup \exists P.D, \exists P.D\}$$

Adding $\exists P.D$ brings the semantics that a must be related to some anonymous x (which must be a member of D) by P; therefore, we can continue to update the tableau as:

$$\begin{array}{ll} \circ\, a & L(a) = \{C, \forall P.(\neg E), \neg C \sqcup \exists P.D, \exists P.D\} \\ \quad\downarrow P & \\ \circ\, x & L(x) = \{D\}, \quad L(a, x) = \{P\} \end{array}$$

where there is a new node x representing x, and a direct edge P from a to x which represents P. Moreover, $L(x) = \{D\}$ and $L(a, x) = \{P\}$ is included in the tableau, as x must be an instance of D, and P relates a to x.

Afterwards, we consider the T-Box axiom $\neg D \sqcup E$ in $O_{\mathcal{ALC}} \cup \neg(\exists P.E)(a)$, the axiom can be added to $L(x)$ which is consequently updated to $L(x) = \{D, \neg D \sqcup E\}$. Again, $\neg D \sqcup E$ denotes a disjunction, we have two options of adding $\neg D$ or E to $L(x)$. However, the first option of adding $\neg D$ contradicts D which is already in $L(x)$, and we move to the second option of adding E. Hence, the tableau constructed so far is displayed as:

$$\begin{array}{ll} \circ\, a & L(a) = \{C, \forall P.(\neg E), \neg C \sqcup \exists P.D, \exists P.D\} \\ \quad\downarrow P & \\ \circ\, x & L(x) = \{D, \neg D \sqcup E, E\}, \quad L(a, x) = \{P\} \end{array}$$

Here, notice that $\forall P.(\neg E)$ in $L(a)$ expresses the semantics that every individual (including x)

to which a is related by P must be a member of $\neg E$. Thus, $\neg E$ has to be added to $L(x)$, which contradicts E already in $L(x)$. This contradiction is not avoidable, as we do not have another option (i.e. the tableau construction terminates with a contradiction). Back to the inference task of checking that whether $O_{\mathcal{ALC}} \models (\exists P.E)(a)$ or not, since $O_{\mathcal{ALC}} \cup \neg(\exists P.E)(a)$ is checked as inconsistent, we will obtain the A-Box fact $(\exists P.E)(a)$ as a valid consequence, i.e. $O_{\mathcal{ALC}} \models (\exists P.E)(a)$.

To sum up the tableaux algorithm, it reduces each inference task to the problem of consistency checking, which is performed by constructing a tableau. The tableau construction is failed if an unavoidable contradiction happens; otherwise, the construction succeeds. Here we list in Figure 3.1 how the tableau is built from all $\mathcal{ALC}$ expressions, whereas a complete introduction considering $\mathcal{SROIQ}$ DL can be found in [HKR09].

- T-Box Axioms C_E: if $C_E \notin L(x)$, then add C_E into $L(x)$
- $C_{E_1} \sqcap C_{E_2}$: if $C_{E_1} \sqcap C_{E_2} \in L(x)$ and $\{C_{E_1}, C_{E_2}\} \not\subseteq L(x)$, then add both C_{E_1} and C_{E_2} into $L(x)$
- $C_{E_1} \sqcup C_{E_2}$: if $C_{E_1} \sqcup C_{E_2} \in L(x)$ and $\{C_{E_1}, C_{E_2}\} \cap L(x) = \emptyset$, then add C_{E_1} or C_{E_2} into $L(x)$
- $\exists P.C_E$: if $\exists P.C_E \in L(x)$ and there is no such an individual y that $P \in L(x, y)$ and $C_E \in L(y)$, then:
  1. add a new node y representing y
  2. add P into $L(x, y)$
  3. add C_E into $L(y)$
- $\forall P.C_E$: if $\forall P.C_E \in L(x)$ and there is a node y with $P \in L(x, y)$ and $C_E \notin L(y)$, then add C_E into $L(y)$

Figure 3.1: Handing $\sqcap$, $\sqcup$, $\forall$ and $\exists$ by tableaux algorithm

3.4 Rule-based Inference

Tableaux-based reasoners are known to provide sound and complete inference, but they often become inefficient when handling large A-Boxes [MS06]. Take the inference task *instance retrieval* as an example, the tableaux algorithm will conduct an *instance checking* for every

individual and every OWL class. This is intractable when ontologies have a large number of A-Box facts; therefore, large reasoners usually adopt another inference technique called rule-based inference, where certain inference tasks can be achieved by applying some reasoning rules over bulk of A-Box facts, rather than each of them individually. Rule-based inference often requires to sacrifice the full expressiveness of OWL 2 for some desirable computational properties. In this section, we first review the three OWL 2 profiles, especially the QL and RL profiles, which aim to support applications considering large A-Boxes. Then, we illustrate query-rewriting approaches and materialised approaches, which are two major categories of rule-based inference.

3.4.1 OWL 2 Profiles

The most recent OWL 2 release has more expressive power compared to its predecessor, OWL 1, because it introduces some new features to represent more complex relations and constraints. Consequently, OWL 2 ontologies require more complex inference. However, in some circumstances, applications might not need the full expressive power of OWL 2. and they might focus only on some of the inference tasks reviewed in Section 3.2. Therefore, OWL 2 further provides OWL 2 EL, OWL 2 QL and OWL 2 RL as its three sub-languages, each of which targets some particular applications, and guarantees tractable inference for some inference tasks.

In particular, the EL profile benefits applications which deal with ontologies that have numerous classes and/or properties. The QL profile targets applications whose major inference task is query processing over large A-Boxes of data. Finally, the RL profile also aims for applications which deal with ontologies having large volumes of data, over which scalable inference is particularly required. In this review, we omit the details of OWL 2 EL, because the thesis focuses on ontologies with a small and static T-Box (but with large A-Boxes).

OWL 2 QL

OWL 2 QL is introduced mainly for applications which handle ontologies with large-scale instance data, and in which query processing is the most important inference task. By using

a suitable inference technique, sound and complete conjunctive query processing over OWL 2 QL ontologies can be performed in LOGSPACE time w.r.t. the size of data (i.e. A-Box). The logical underpinning of OWL 2 QL is DL-Lite$_R$ [CDGL$^+$07] (a DL fragment), where the use of SameIndividual is not supported, and thus practical inference problems related to not using the UNA are avoided. OWL 2 QL can be extended with features from other members of DL-Lite family, such as DL-Lite$_A$ (which extends DL-Lite$_R$ with FunctionalProperty), but the UNA has to be additionally adopted in order to guarantee LOGSPACE time query processing.

This OWL 2 QL profile plays an important role in the topic of **Ontology-based Data Access** (**OBDA**) [PK03]. In OBDA explicit facts in A-Boxes are stored in a database (e.g. an RDBMS); then T-Box axioms in an OWL 2 QL ontology are used to rewrite a query to some sub-queries, which computes implicit answers (not materialised) alongside explicit answers (materialised) to the original query. This strategy does not materialise the implicit answers, but dynamically processes queries by rewriting them, which is known as **query-writing** or backward-chaining inference [UVHSB11].

OWL 2 QL restricts the use of the full OWL 2 by not only specifying the set of supported expressions, but also the positions of an axiom in which these expressions are allowed to occur. In Table 3.3, we summarise for OWL 2 QL the class and property expressions it supports, and restrictions depending on their positions in axioms of the DL form SubExpression $\sqsubseteq$ SuperExpression (or LeftHandExpression $\sqsubseteq$ RightHandExpression). For instance, some constructors such as UnionOf are totally disallowed in order to avoid non-deterministic inference. Moreover, when using SomeValuesFrom to form a subclass expression, the existential restriction has to be unqualified (i.e. $\exists P_E$). Also when using the constructor IntersectionOf to form a super-class expression; the class expressions (i.e. C_{E_1} or C_{E_2} in $C_{E_1} \sqcap C_{E_2}$) concatenated by the constructor must also be expressions supported on the right-hand of QL (denoted as C_{RE_1} and C_{RE_2}).

Because of these restrictions, only limited axioms listed in Table 3.4 are expressible in OWL 2 QL ontologies. For example, since the UnionOf constructor is not allowed, specifying the axiom of $C_{E_1} \sqcup C_{E_2} \sqsubseteq C_{E_3}$ is not possible. In summary, the QL profile does not support axioms FunctionalProperty, InverseFunctionalProperty, IrreflexiveProperty, TransitiveProperty and

Table 3.3: Class & property expressions and restrictions in OWL 2 QL and OWL 2 RL

	Exp. Constructor	DL Syntax	OWL 2 QL		OWL 2 RL	
			Sub-Exp.	Super-Exp.	Sub-Exp.	Super-Exp.
Class Exp.	Thing	$\top$	✓	✓	✗	✗
	Nothing	$\bot$	✓	✓	✓	✓
	AtomicClass	C	✓	✓	✓	✓
	IntersectionOf	$C_{E_1} \sqcap C_{E_2}$	✗	$C_{RE_1} \sqcap C_{RE_2}$	$C_{LE_1} \sqcap C_{LE_2}$	$C_{RE_1} \sqcap C_{RE_2}$
	UnionOf	$C_{E_1} \sqcup C_{E_2}$	✗	✗	$C_{LE_1} \sqcup C_{LE_2}$	✗
	ComplementOf	$\neg C_E$	✗	$\neg C_{LE}$	✗	$\neg C_{LE}$
	AllValuesFrom	$\forall P_E.C_E$	✗	✗	✗	$\forall P_E.C_{RE}$
	SomeValuesFrom	$\exists P_E$	✓	✓	✓	✗
		$\exists P_E.C_E$	$\exists P_E.\top$	$\exists P_E.C_{RE}$	$\exists P_E.C_{LE}$	✗
	HasSelf	$\exists P_E.\mathsf{Self}$	✗	✗	✓	✓
	HasValue	$\exists P_E.\{a\}$	✗	✗	✓	✓
	MaxCardinality	$\leqslant n\, P_E$	✗	✗	✗	$\leqslant 1\, P_E$
		$\leqslant n\, P_E.C_E$	✗	✗	✗	$\leqslant 1\, P_E.C_{LE}$
	OneOf	$\{a_1 \ldots a_n\}$	✗	✗	✓	✗
Property Exp.	AtomicProperty	P_A	✓	✓	✓	✓
	InverseOf	P_A^-	✓	✓	✓	✓

PropertyChain, and HasKey. With regard to assertions, the use of SameIndividual is forbidden in order to obey the UNA; furthermore, using NegativePropertyAssertion is disallowed.

OWL 2 RL

OWL 2 RL sacrifices some expressivity of the OWL 2 for inference scalability. The RL profile is inspired from the **Description Logic Program** (**DLP**) [GHVD03] and pD* [tH05], and it enables applications to perform inference through a forward-chaining mechanism [KWE10]. OWL 2 RL essentially supports all constructors of OWL 2 (except ReflexiveProperty and DisjointUnion), but restricts the use of them in a syntactic manner (i.e. in axioms of the form SubExpression $\sqsubseteq$ SuperExpression, the supported constructors in the left-hand expressions are different from the supported ones which are allowed in the right-hand expressions). Again, we list in Table 3.3 and Table 3.4 the detailed restrictions in OWL 2 RL of using constructors and supported axioms, respectively. In general, the RL profile does not allow the use of the class Thing (i.e. $\top$), and thus the constructor ReflexiveProperty is not permitted. Moreover, UnionOf can only be used to construct a subclass but not a super class, in order to avoid non-deterministic inference, so that the use of DisjointUnion is disallowed. Unlike the QL profile,

Table 3.4: Axioms supported in OWL 2 QL and OWL 2 RL

	Axiom Constructor	DL Syntax	OWL 2 QL	OWL 2 RL
Class Axioms	SubClassOf	$C_{E_1} \sqsubseteq C_{E_2}$	$C_{LE} \sqsubseteq C_{RE}$	$C_{LE} \sqsubseteq C_{RE}$
	EquivalentClasses	$C_{E_1} \equiv C_{E_2}$	$C_{LE_1} \equiv C_{LE_2}$	✓
	DisjointClasses	$\mathsf{DisCla}(C_{E_1} \dots C_{E_n})$	$\mathsf{DisCla}(C_{LE_1} \dots C_{LE_n})$	$\mathsf{DisCla}(C_{LE_1} \dots C_{LE_n})$
	UnionOf	$C_{E_1} \sqcup C_{E_2} \sqsubseteq C_{E_3}$	✗	$C_{LE_1} \sqcup C_{LE_2} \sqsubseteq C_{RE}$
	DisjointUnion	$\mathsf{DisUni}(C, C_{E_1} \dots C_{E_n})$	✗	✗
Property Axioms	SubPropertyOf	$P_{E_1} \sqsubseteq P_{E_2}$	✓	✓
	EquivalentProperties	$P_{E_1} \equiv P_{E_2}$	✓	✓
	DisjointProperties	$\mathsf{DisPro}(P_{E_1} \dots P_{E_n})$	✓	✓
	InverseProperties	$P_{E_1} \equiv P_{E_2}^-$	✓	✓
	PropertyDomain	$\mathsf{Dom}(P_E, C_E)$	$\mathsf{Dom}(P_E, C_{RE})$	$\mathsf{Dom}(P_E, C_{RE})$
	PropertyRange	$\mathsf{Rng}(P_E, C_E)$	$\mathsf{Rng}(P_E, C_{RE})$	$\mathsf{Rng}(P_E, C_{RE})$
	FunctionalProperty	$\mathsf{Fun}(P_E)$	✗	✓
	InverseFunctionalProperty	$\mathsf{InvFun}(P_E)$	✗	✓
	ReflexiveProperty	$\mathsf{Ref}(P_E)$	✓	✗
	IrreflexiveProperty	$\mathsf{Irr}(P_E)$	✗	✓
	SymmetricProperty	$\mathsf{Sym}(P_E)$	✓	✓
	AsymmetricProperty	$\mathsf{Asy}(P_E)$	✓	✓
	TransitiveProperty	$\mathsf{Tra}(P_E)$	✗	✓
	PropertyChain	$P_{E_1} \circ \dots \circ P_{E_1} \sqsubseteq P_E$	✗	✓
	HasKey	$\mathsf{HasKey}(C_E, P_{E_1} \dots P_{E_n})$	✗	$\mathsf{HasKey}(C_{LE}, P_{E_1} \dots P_{E_n})$
Assertions	ClassAssertion	$C_E(a)$	✓	$C_{RE}(a)$
	PropertyAssertion	$P_E(a_1, a_2)$	✓	✓
	SameIndividual	$=(a_1 \dots a_2)$	✗	✓
	DifferentIndividuals	$\neq(a_1 \dots a_2)$	✓	✓
	NegativePropertyAssertion	$\neg P_E(a_1, a_2)$	✗	✓

the RL profile does not allow SomeValuesFrom to construct the super-class expression, which prevents the RL profile from deriving anonymous individuals. With regard to universal quantifications, the RL profile only allows the use of AllValuesFrom to form a super-class expression, but not in a subclass expression. Note that QL and RL profiles overlap each other (i.e. RL $\not\subseteq$ QL and QL $\not\subseteq$ RL).

A key feature of OWL 2 RL is that ontological axioms expressed in this profile can be transformed into logical implications; for instance, the axiom $\mathsf{Rng}(P_E, C_{RE})$ can be transformed as a logical implication $P_E(x, y) \rightarrow C_{RE}(y)$, where variables x and y are universally quantified. Realisation and instance retrieval are the most important inference tasks considered in this profile, and can be performed by implementing logical implications to entailment rules, which are used to derive and materialise valid conclusions (i.e. inferred information). As compared to

query-rewriting approaches, the inference results are computed and materialised before executing a query (which is called a **materialised** approach), both explicit and implicit information can be simply viewed by queries without a rewriting process.

3.4.2 Query-rewriting Approach

A query-rewriting approach usually considers OWL 2 QL ontologies, over which sound and complete query processing can be provided (more precisely answering the **conjunctive queries**) in LOGSPACE time. In contrast to a materialised approach, query-rewriting starts from a query executed over the ontological data, and by using the axioms contained in the ontology, the query is rewritten to a set of sub-queries, which return not only explicitly stored data, but also implicitly derivations as answers to the query. As a result, there is no need to store the inference consequences in query-rewriting, so that it requires less space than materialising all inference results. In addition, the sub-queries which are rewritten from the original query can be expressed in SQL, which enables existing RDBMSs to answer queries with ontological augmentation. However, as queries are processed dynamically, this approach often becomes inefficient in the case that original queries are complex (i.e. queries need to be rewritten to many sub-queries, which makes computation of query answers very difficult and time-consuming) or the case that certain queries are executed frequently and repeatedly.

The simplest form of a query can be specified by replacing the individual names of A-Box assertion with variables (e.g. $\mathsf{Beer}(x)$ or $\mathsf{hasColour}(x, y)$), which retrieves for all instances of a class or property (query processing is closely related to the inference task of instance retrieval). This simplest form is called a **query atom**, and a conjunctive query can be constructed by conjunctively connecting several query atoms. For example, the conjunctive query $\mathsf{Beer}(x) \wedge \mathsf{hasColour}(x, \text{'Dark'})$ searches for beer individuals which have a dark colour. Note that, we can replace some variables in query atoms with constant values (i.e. $\mathsf{Beer}(x) \wedge \mathsf{hasColour}(x, \text{'Dark'})$ logically means $\mathsf{Beer}(x) \wedge \mathsf{hasColour}(x, y) \wedge y{=}\text{'Dark'}$). If we assume, instances of Beer and hasColour are stored in two tables $\mathsf{Beer}(id)$ and $\mathsf{hasColour}(domain, range)$, respectively. This conjunctive query can be intuitively translated as the following SQL query:

```
SELECT id
FROM Beer JOIN hasColour
ON Beer.id = hasColour.domain
AND hasColour.range = 'Dark'
```

Inference systems adopting a query-rewriting approach include DLDB, Ontop and Stardog. In order to demonstrate the process of rewriting queries, we take the inference system DLDB as an example, and consider the axioms (2.24), (2.25) and (2.28). DLDB stores the A-Box facts of an ontology inside an RDBMS, and it uses views as a method to rewrite executed queries. For example, to answer queries retrieving instances of the class Beer, DLDB translates the three axioms into the following views (expressed in a Datalog style):

$$\mathsf{Beer}(x) \text{ :- } \mathsf{Beer}_e(x) \qquad \mathsf{Beer}_i(x) \text{ :- } \mathsf{Ale}(x)$$

$$\mathsf{Beer}(x) \text{ :- } \mathsf{Beer}_i(x) \qquad \mathsf{Beer}_i(x) \text{ :- } \mathsf{Lager}(x)$$

$$\mathsf{Beer}_i(x) \text{ :- } \mathsf{LiquidBread}(x)$$

As can be seen, the view Beer (representing the class Beer) includes both extensional (i.e. explicit facts) and intensional (i.e. inferred facts) sets which are denoted as $\mathsf{Beer}_e(x)$ and $\mathsf{Beer}_i(x)$, respectively, where the intensional one consists of data taken from views $\mathsf{Ale}(x)$, $\mathsf{Lager}(x)$ and $\mathsf{LiquidBread}(x)$. By using the views, any query retrieving instances of the class Beer will include not only explicitly asserted facts, but also derived assertions. Since inside an RDBMS, views are often used as the mechanism for rewriting queries, systems like DLDB and Ontop can be alternatively classified as view-based inference [MRS12]. Continuing with the example, now for only considering the three axioms, the view of LiquidBread is:

$$\mathsf{LiquidBread}(x) \text{ :- } \mathsf{LiquidBread}_e(x) \qquad \mathsf{LiquidBread}_i(x) \text{ :- } \mathsf{Beer}(x)$$

$$\mathsf{LiquidBread}(x) \text{ :- } \mathsf{LiquidBread}_i(x)$$

where the intensional view LiquidBread contains the view $\mathsf{Beer}(x)$. Note that, in query-rewriting approaches, there might exist mutual recursions (e.g. the above view definitions $\mathsf{Beer}(x)$ and $\mathsf{LiquidBread}(x)$ recursively include the other).

3.4.3 Materialised Approach

A materialised approach usually focuses on (but is not limited to) OWL 2 RL ontologies, and considers realisation and instance retrieval (which can be more generally interpreted as type inference as we have defined) as the most significant inference tasks. Inference systems which adopt this approach include Oracle RDF Store, OWLim, Minerva, RDFox, WebPIE, DynamiTE [UMJ+13], Cichlid and SQOWL. Axioms in the RL profile can be interpreted as logical implications, which are translated into entailment rules by a materialised approach. The entailment rules are then applied to the ontologies, in order to compute and materialise the memberships of class and property instances. As a consequence, any query on the ontology is directly executed over the materialised facts, so that dynamically computing the answers to queries is no longer required. This benefits applications where queries are asked frequently. However, since a materialised approach stores both explicit and implicit facts for a given ontology, it requires more space than query-rewriting approaches. Moreover, if the data of the ontology is often updated, how the materialisation can be incrementally updated raises another challenge for materialised approaches.

Entailment rules can be written in the form **if** $\langle premise \rangle$ **then** $\langle conclusion \rangle$, which captures the semantics (i.e. logical implications) of OWL 2 RL axioms. Take the PropertyRange axiom (denoted as $\mathsf{Rng}(P_E, C_{RE})$ in DL) as an example, the logical implication implies if an individual x is related to y by P_E, then y must be inferred as an instance of the class expression C_{RE}; consequently, an entailment rule can be obtained as:

$$\textbf{if } P_E(x, y) \textbf{ then } C_{RE}(y)$$

Note that the free variables x and y in the field of $\langle premise \rangle$ are universally quantified. Instantiating this entailment rule over the axiom (2.33), which defines the PropertyRange of the property hasFlavour, results in an entailment rule:

$$\textbf{if } \mathsf{hasFlavour}(x, y) \textbf{ then } \mathsf{Flavour}(y)$$

Therefore, all object values contained in hasFlavour will be inferred as members of the class Flavour. Thus, through a materialised approach, many instance memberships can be computed

at once, which is more efficient than the task of instance checking, which checks for each instance individually.

An instantiated rule is **applicable** to an ontology if the ontology has instances matching the respective $\langle premise \rangle$ of the rule, and as a result, the $\langle conclusion \rangle$ can be added as implicit derivations to the ontology. Thus checking which entailment rules are applicable vitally influences the performance of an inference system using a materialised approach. The RL profile comes with a set of entailment rules, called OWL 2 RL/RDF rules, and a materialised approach can instantiate them, and then apply the instantiated rules over the data of the ontology, so that implicit information can be derived and materialised.

Note that, a materialised approach should guarantee that the order of applying the set of entailment rules does not change the overall inference result, if there are multiple choices of applying these rules. This requires that in the set of entailment rules, applying one of them does not prevent the application of any others. Such a property is formally called **monotonicity** [KSH12], which holds for the set of OWL 2 RL/RDF rules, and an inference system providing this property is then called a monotone inference system.

3.5 Summary

In this chapter, we have introduced inference tasks commonly considered in OWL 2, and reviewed two types of inference methods, which are tableaux-based inference and rule-based inference. Tableaux-based inference relies on the tableaux algorithm which first reduces all inference tasks to the problem of consistency checking, and then solves the inference tasks through a tableau construction process. Tableaux algorithm is known to provide sound and complete inference for ontologies expressed in $\mathcal{SROIQ}$. However, tableaux-based inference suffers an inefficiency problem when ontologies have large A-Boxes.

An alternative approach called rule-based inference provides tractable inference. Rule-based inference performs inference by using a set of reasoning rules. OWL 2 provides three profiles, in which the QL and RL profiles focusing on handling large volumes of data, which is a focus

of the work in this thesis. The two profiles restrict the use of the full OWL 2 in different ways, in order to support certain applications with desired computational properties. Applications which care most about query processing might adopt a query-rewriting approach and focus more on the OWL 2 QL ontologies, and conjunctive query processing can be performed in LOGSPACE time. In a query-rewriting approach, queries over ontologies are rewritten to some sub-queries, which dynamically compute but do not materialise the inferred results. This differentiates query-rewriting from the materialised approach, that computes and materialises inference results before executing a query, especially for ontologies in the RL profile, which aims for scalable inference. A query-rewriting approach requires less space than a materialised approach, but it will be inefficient when queries are difficult to rewrite or have been executed repeatedly.

If we compare tableaux-based with rule-based inference, the former definitely handles more DL constructs than the latter, since the tableaux algorithm covers all $\mathcal{SROIQ}$. Rule-based inference usually provides only sound inference due to the separation of T-Box and A-Box, as the data size considered is often very large (which makes considering T-Box and A-Box together impossible in practice). A detailed analysis of incomplete inference because of this separation is provided in Section 5.6 of Chapter 5. Thus rule-based inference usually considers ontologies expressed in a restricted sub-language of OWL 2. Note that, our approach focuses inference in databases, where schemas are often simple and static compared to the large and dynamic base data, and queries are more frequent than updates. This leads our approach to taking a materialised approach.

Chapter 4

Type Inference from Data Inserts

4.1 Introduction

Type inference derives for each instance its membership of classes or properties. It differs from the inference task **instance retrieval** (which we described in Section 3.2.2 of Chapter 3) on it not only derives the memberships of class instances, but also of property instances. Type inference systems over large ontologies can be classified into query-rewriting and materialised approaches, as shown in Figure 4.1. Systems using query-rewriting (e.g. DLDB, Stardog, Ontop and Lutz et al. [LTW09]) store only explicit facts and conduct inference only when there is a query executed over the ontology. A consequence of using query-rewriting is that no incremental type inference is required. DLDB, Ontop and Lutz et al. store data inside an RDBMS, and can be considered as transactional reasoning systems, as they use views inside the RDBMS as the method to rewrite executed queries. By contrast, Stardog is not an RDBMS-based system, and therefore provides non-transactional reasoning.

Type inference systems based on materialised approaches store both explicit and implicit data, in order to provide a fast query processing service [MRS12]. Most materialisation-based systems perform inference outside of an RDBMS, and this offer non-transactional reasoning, even in cases where they still choose an RDBMS as the data container. WebPIE, as an example inference engine, applies the MapReduce model to build the inference mechanism for RDF semantics [HPS14] and OWL ter Horst semantics [tH05] on top of a Hadoop cluster. WebPIE

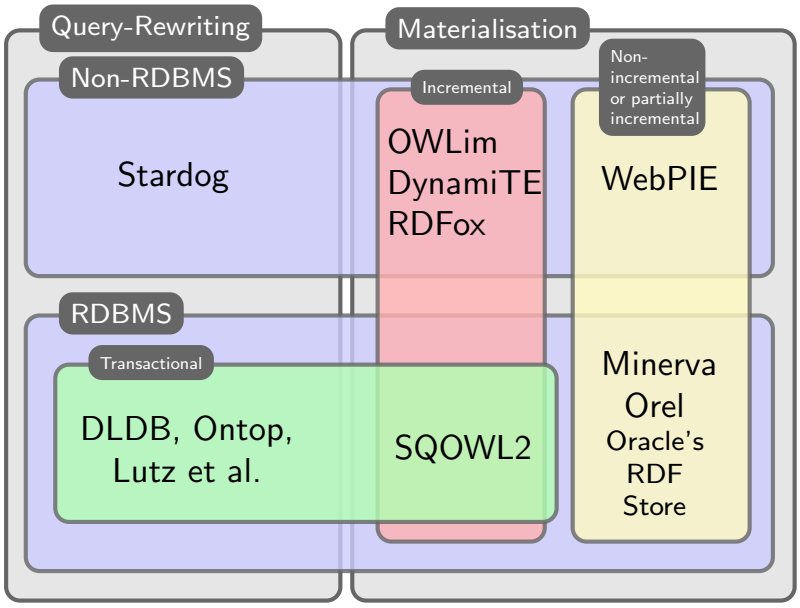

Figure 4.1: Categories of Type Inference Systems

only supports incremental data loading, but not incremental deleting. OWLim is another triple-store system, which uses a file system instead of an RDBMS as a container for storing semantic data. Its standard and enterprise versions support incremental data loading and deleting, but not in a transactional manner. RDFox adopts a so-called Backward/Forward algorithm [MNPH15] (can be more efficient than DRed in certain cases) to achieve incremental inference without using an RDBMS, and it is, therefore, not a transactional reasoning system. DynamiTE is another system handling both incremental inserts and deletes w.r.t. the minimal RDFS reasoning [MVPG09], by using proposed variants of the Counting and DRed algorithms in [GMS93]. Again, it does not adopt an RDBMS as its mechanism of inference computation and closure storage. Minerva only uses an RDBMS to hold the materialised results generated by an extra reasoner outside the RDBMS, so we do not consider it as a transactional or incremental inference system. Oracle's RDF store, by contrast, loads the explicit data in advance, and then uses inference rules to generate an inference closure of the loaded data. Although the inference is inside the database, it is not performed in an incremental manner. Similar to Oracle's RDF store, Orel [KMR10] also uses a relational database to compute and store an inference closure, and their inference rules address the EL and the RL profiles of OWL 2.

This chapter focuses on type inference in an RDBMS, and this part of work is called SQOWL2, which we believe is unique in providing transactional reasoning combined with materialisation

of the inferred data. In our approach, we adopt a tableaux-based reasoner for the **classification** over the T-Box of an ontology, and then we used the classified T-Box to build an **ATIDB**, which contains tables and triggers. When A-Box facts are loaded into or removed from the ATIDB, the inserts and deletes cause triggers to be invoked, which performs type inference in a **transactional** and **incremental** manner. Note that as a materialised approach, we mainly focus on axioms expressed in OWL 2 RL, but we also analyse some cases (i.e. using SomeValuesFrom to form a superclass expression, MinCardinality, ReflexiveProperty and SelfRestriction) which are beyond that profile, in order to demonstrate that our approach is not limited to OWL 2 RL.

SQOWL2 extends the previous work SQOWL supporting type inference for OWL 1 only after inserts were made to an RDBMS, to now perform type inference for OWL 2 after both inserts and deletes. In this chapter, we focus on type inference from data inserts, while handling deletes is provided in Chapter 5. We first use a motivating example to illustrate what problems might occur in type inference in Section 4.2. Then, Section 4.3 briefly introduces the overview of the approach. Section 4.4 describes how SQOWL2 establishes for a given ontology a canonical schema, where triggers can be created. Section 4.5 then details how the triggers are generated from OWL 2 axioms. In Section 4.6, we provide some optimisations that can be applied to further improve SQOWL2, and finally Section 4.7 summarises this chapter.

4.2 Motivating Example

Type inference is very similar to the DL inference task **instance retrieval**, which computes all individuals contained in a class. Besides computing all instances of classes, type inference also derives all instances of properties. Thus, type inference plays an important role when querying the information in an ontology. To address the problem of type inference, we consider again the axiom (1.1) in the beer ontology.

$$\mathsf{CzechLager} \equiv \mathsf{Lager} \sqcap \mathsf{CzechBeer} \tag{1.1}$$

which expresses that a CzechLager is something that is both a Lager and a CzechBeer. This axiom can be replaced during the classification with the three SubClassOf axioms (1.5) – (1.7)

by a tableaux-based reasoner, such as Pellet, Hermit and FaCT++ [TH06].

$$\mathsf{CzechLager} \sqsubseteq \mathsf{Lager} \quad (1.5)$$

$$\mathsf{CzechLager} \sqsubseteq \mathsf{CzechBeer} \quad (1.6)$$

$$\mathsf{Lager} \sqcap \mathsf{CzechBeer} \sqsubseteq \mathsf{CzechLager} \quad (1.7)$$

$$\mathsf{Lager}(\mathsf{CzechPaleLager}) \quad (1.2)$$

$$\mathsf{CzechBeer}(\mathsf{CzechPaleLager}) \quad (1.3)$$

$$\mathsf{CzechLager}(\mathsf{CzechDarkLager}) \quad (1.4)$$

where (1.5) denotes every CzechLager is a Lager, (1.6) denotes every CzechLager is a CzechBeer, and finally (1.7) represents that some beer which is both a Lager and a CzechBeer is a CzechLager. Suppose individuals from the three classes are stored in tables CzechLager, Lager and CzechBeer in an RDBMS, and CzechPaleLager as a Lager specified by the above A-Box fact (1.2) has been already recorded in the database shown as S_0 in Figure 4.2.

Figure 4.2: $T_{4.1}$: inserting CzechBeer(CzechPaleLager)

If a database transaction $T_{4.1}$ inserts the above A-Box fact (1.3), then CzechPaleLager should be viewed not only in tables Lager and CzechBeer, but also in CzechLager because of (1.7) (i.e. CzechPaleLager should be inferred as a CzechLager), and the database should be updated to S_1. Now we further consider another transaction $T_{4.2}$ which loads (1.4) as a database insert, the expected change of the database is illustrated in Figure 4.3.

Figure 4.3: $T_{4.2}$: inserting CzechLager(CzechDarkLager)

As a result of type inference, CzechDarkLager should be viewed as not only a CzechLager but also a Lager and CzechBeer because of (1.5) and (1.6) (i.e. the database should be incrementally updated from S_1 to S_2).

However, many approaches which store ontological data in an RDBMS, such as Oracle's RDF Store and Minerva, will make the inference be detached from transaction processing of the data. Take $T_{4.1}$ inserting (1.3) as an example, after CzechBeer(CzechPaleLager) has been inserted, these approaches need to perform a separate inference process to infer CzechPaleLager as a CzechLager. Thus, they fail to provide what has been termed as **transactional reasoning**, where inference from data updates by the database transactions should be available as part of the atomic action of the transactions. Considering transactional reasoning in the above example, viewing CzechLager(CzechPaleLager) should be an atomic result of $T_{4.1}$ rather than requiring an additional inference process. In other words, any other transaction T_c simultaneously executed with $T_{4.1}$ should view the database as either S_0 or S_1 but not any other intermediate state (i.e. the statement CzechBeer(CzechPaleLager) $\sqcap$ $\neg$CzechLager(CzechPaleLager) should always evaluate as false).

Performing type inference from data deletes is a significantly more complex problem. For instance, for a fact that is implicitly inferred, deleting it might cause the database to be inconsistent. In the above example, in the database state S_1 or S_2, deleting the derived fact CzechLager(CzechPaleLager) contradicts what can be inferred from (1.7), (1.2) and (1.3); therefore, attempting such a delete should be rejected. Furthermore, in S_2, removing CzechDarkLager as a CzechLager (i.e. deleting the A-Box fact (1.4)) might appear to lead to a view update problem [Dat00], i.e. which one of Lager and CzechBeer should CzechDarkLager be deleted from? Due to the difficulties, most inference systems do not handle deletes in an incremental manner. For example, the Lite version of OWLim needs to re-compute the whole inference closure every time there is a delete instead of incrementally updating the existing closure. SQOWL2 for performing type inference from data deletes is detailed in Chapter 5, and in this chapter we focus on how SQOWL2 builds an ATIDB for handling data inserts.

4.3 SQOWL2 Overview

In this section, we give an overview of SQOWL2, which provides service of type inference in a transactional and incremental way in an RDBMS. We first introduce the approach architec-

ture, which separates T-Box and A-Box inference. Then, we use the motivating example in Section 4.2 to demonstrate the process of transforming an ontology into tables and triggers, both of which form an ATIDB.

4.3.1 SQOWL2 Architecture

As shown in Figure 4.4, SQOWL2 follows a materialised inference architecture which separates T-Box and A-Box inference. Although this separation might result in incomplete reasoning [Krö12b] (which we will exemplify in Section 5.6), it is not abnormal in other reasoners, such as DLDB and Minerva, which perform inference over large A-Boxes. For the first phase of T-Box inference, we use a tableaux-based reasoner due to its complete classification inherited from the tableaux algorithm. Examples of such reasoners include Pellet, Hermit and FaCT++, and a detailed comparison of them is provided in [Abb12]. We choose Pellet from them, because it is well-documented, and known to provide sound and complete reasoning, but there is no restriction for us to use other tableaux-based reasoners. The classification gives us complete subsumption relations w.r.t. the T-Box, and such a classified T-Box is used for building the ATIDB in three steps:

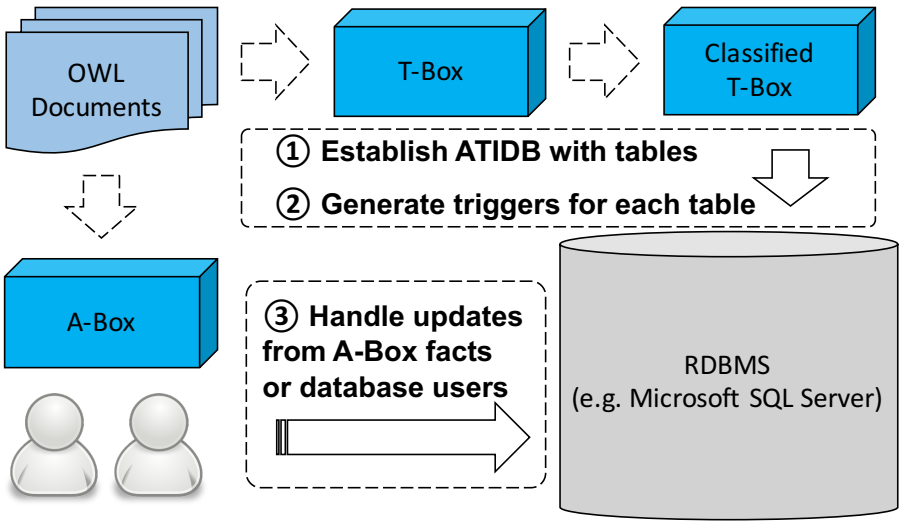

Figure 4.4: Approach Architecture for Type Inference in an RDBMS

1. Classes and properties, as unary and binary relations, are transformed into tables with one and two columns, respectively (i.e. $C \rightsquigarrow C(id)$ and $P \rightsquigarrow P(domain, range)$, where

C and P denote class and property tables, respectively). In our motivating example, the three classes CzechLager, CzechBeer and Lager are mapped to three tables CzechLager(id), CzechBeer(id) and Lager(id), respectively. These tables representing classes and properties form the basic schema of the ATIDB.

2. SQOWL2 creates triggers for each table, based on the axioms related to the class or property, which the table represents. In this process, the semantics of the axioms are captured by the triggers. Here, we only demonstrate triggers in a logical manner by using ECA rules of the form **when** ⟨event⟩ **if** ⟨condition⟩ **then** ⟨action⟩. We call triggers expressed in such a logical way **logical triggers**, which can be implemented as **physical triggers** in different RDBMSs, such as Postgres or Microsoft SQL Server.

3. The established ATIDB is ready to accept data updates (inserts especially in this chapter) either translated from A-Box facts, or directly from database users. When there is an update executed over a table, triggers associated with this table will be automatically invoked to perform the process of type inference. Trigger invoking naturally preserves the ACID properties in an RDBMS, and consequently, SQOWL2 supports transactional reasoning.

Here, it is worth mentioning that, we only use tableaux-based reasoners for classifying the T-Boxes of which the size is suitable for tractable classification (i.e. fewer than hundreds of classes, properties and axioms are contained in a T-Box), and consequently benefit from their complete T-Box inference. When dealing with very large T-Boxes, such as SNOMED-CT [Don06], SQOWL2 can be adjusted to use other types of reasoners, such as [DK09] and TrOWL [TPR10] (the former uses an RDBMS to classify ontologies based on a simpler $\mathcal{ELH}$ DL fragment, and the latter offers tractable support for all the expressive power of OWL 2).

4.3.2 Type Inference from Data Inserts by Triggers

We use the motivating example (i.e. the T-Box axiom (1.1) and A-Box facts (1.2), (1.3) and (1.4)) to demonstrate the above three steps of establishing an ATIDB. As we have already

introduced, (1.1) is classified as three new SubClassOf axioms (1.5) – (1.7), and classes which appear in these axioms are represented as tables CzechLager(id), CzechBeer(id) and Lager(id). For the classified axiom (1.5), asserting any fact x about a CzechLager, implies that x is also a Lager, which we implement as the trigger:

$$\textbf{when } {}^{+}\mathsf{CzechLager}(x) \textbf{ then } \mathsf{Lager}(x)$$

The "+" symbol before CzechLager(x) indicates that this is an **after trigger**, which in SQL means something that is executed after the datum x has been put into a table. More specifically, the above trigger means that after x has been inserted into the table CzechLager, another insertion will be generated by the trigger to insert the same x into the table Lager. Very similarly for the axiom (1.6), we implement another trigger:

$$\textbf{when } {}^{+}\mathsf{CzechLager}(x) \textbf{ then } \mathsf{CzechBeer}(x)$$

which will be invoked to insert x to the table CzechBeer, after the same x has been put into CzechLager. However, the axiom (1.7) is more complex as it expresses a subsumption relation between the IntersectionOf expression Lager $\sqcap$ CzechBeer and the atomic class CzechLager, and when either Lager or CzechBeer is asserted for including a fact x, the existence of the same fact x in the other class must be checked before we can assert that x is a CzechLager. Thus, this axiom creates an after trigger for each of the tables Lager and CzechBeer as follows:

$$\textbf{when } {}^{+}\mathsf{Lager}(x) \textbf{ if } \mathsf{CzechBeer}(x) \textbf{ then } \mathsf{CzechLager}(x)$$

$$\textbf{when } {}^{+}\mathsf{CzechBeer}(x) \textbf{ if } \mathsf{Lager}(x) \textbf{ then } \mathsf{CzechLager}(x)$$

The insertion of the data x to CzechLager from the above triggers will again fire the two after triggers created on CzechLager. However, this results in infinite loops, since the fact of x as a Lager and a CzechBeer would lead to the fact of x as a CzechLager, which in turn leads to x being asserted as a Lager and a CzechBeer again. To prevent this, SQOWL2 creates for each table a **before trigger** (indicated by the "−" prefix as shown in the below triggers for Lager, CzechBeer and CzechLager), which is invoked before inserting the actual data into the table.

$$\textbf{when } {}^{-}\mathsf{Lager}(x) \textbf{ if } \neg\mathsf{Lager}(x) \textbf{ then } \mathsf{Lager}(x)$$

$$\textbf{when } {}^{-}\mathsf{CzechBeer}(x) \textbf{ if } \neg\mathsf{CzechBeer}(x) \textbf{ then } \mathsf{CzechBeer}(x)$$

$$\textbf{when } {}^{-}\mathsf{CzechLager}(x) \textbf{ if } \neg\mathsf{CzechLager}(x) \textbf{ then } \mathsf{CzechLager}(x)$$

As can be seen, these before triggers validate that the data does not exist before attempting an insert, and consequently duplicate insertions and infinite loops are avoided.

4.3.3 SQOWL2 Features

As shown in Section 4.3.2, triggers are automatically invoked to perform type inference whenever there is an insert executed over tables in the ATIDB. Besides the ability to provide transactional reasoning, which preserves ACID properties, SQOWL2 has the following features:

- As a materialised approach, the materialising inference results might require much more space than non-materialising approaches, such as Ontop and DLDB (especially if there are numerous implicit facts being derived); however, the materialised closure would make query processing much faster, because queries only need to simply read the materialised views rather than compute the answer dynamically,

- Type inference provided by SQOWL2 is conducted in an incremental way, as each new insert captured by triggers causes all additional facts which can be derived from the combination of the new insert and existing data to be added into the ATIDB. Thus, triggers being invoked by new inserts will analyse how the previous inference closure should be incrementally updated because of the new inserts rather than re-computing and materialising the whole inference closure.

- Due to the separation of T-Box and A-Box inference, even by applying a complete reasoner (i.e. a tableaux-based reasoner), we still miss those T-Box axioms that need to be inferred by considering both T-Box and A-Box together. Consequently, the inference provided by SQOWL2 is incomplete. We make a conjecture, demonstrated by later experimental evaluation, that the type inference over OWL 2 RL ontologies is complete.

4.4 Establishing the ATIDB Schema

In Section 4.3, we have already given an overview of SQOWL2 for establishing an ATIDB which is able to perform type inference. The ability of inference by SQOWL2 mainly comes from the triggers translated from OWL 2 axioms. Before we thoroughly demonstrate the trigger generation in Section 4.5, this section first outlines how we establish a relational schema (representing OWL classes and properties), where triggers can be created later.

4.4.1 Canonical Schema

We call the relational schema a **canonical schema**, which contains tables with one or two columns. In the canonical schema, OWL classes which denote unary relations are represented as one-column tables, and OWL properties denoting binary relations are represented as two-column tables. More precisely, for an atomic class C, SQOWL2 creates a table $C(id)$, and the column id is set as the **Primary Key** (**PK**) because we follow the UNA. Similarly, for an atomic property P, we create a table with two columns as $P(domain, range)$, and we set $(domain, range)$ as a compound PK for this table. Thus, building the canonical schema can be summarised as the following two rules (where underlining a column or columns means setting them as a key):

$$C \rightsquigarrow C(\underline{id});\ P \rightsquigarrow P(\underline{domain, range})$$

In addition to the above two rules, for SubClassOf axioms between two atomic classes, and SubPropertyOf axioms between two atomic properties, we add FKs (symbolised as $\overset{\text{fk}}{\Rightarrow}$) detailed in the rules below to represent the semantics of subsumption:

$$\text{SubClassOf} : C \sqsubseteq D \rightsquigarrow C(id) \overset{\text{fk}}{\Rightarrow} D(id)$$

$$\text{SubPropertyOf} : P \sqsubseteq Q \rightsquigarrow P(domain, range) \overset{\text{fk}}{\Rightarrow} Q(domain, range)$$

For instance, if we only consider (2.24) and (2.25), which are two SubClassOf axioms denoting Ale and Lager as two subclasses of Beer, and (2.47) and (2.48), which are two SubPropertyOf axioms denoting hasColour and hasFlavour as two sub properties of hasDescription, the canonical schema is generated as follows. The three classes and three properties are mapped to the

following tables:

$\mathsf{Ale} \rightsquigarrow \mathsf{Ale}(\underline{\mathsf{id}})$ $\quad\quad$ $\mathsf{hasColour} \rightsquigarrow \mathsf{hasColour}(\underline{\mathsf{domain}, \mathsf{range}})$

$\mathsf{Lager} \rightsquigarrow \mathsf{Lager}(\underline{\mathsf{id}})$ $\quad\quad$ $\mathsf{hasFlavour} \rightsquigarrow \mathsf{hasFlavour}(\underline{\mathsf{domain}, \mathsf{range}})$

$\mathsf{Beer} \rightsquigarrow \mathsf{Beer}(\underline{\mathsf{id}})$ $\quad\quad$ $\mathsf{hasDescription} \rightsquigarrow \mathsf{hasDescription}(\underline{\mathsf{domain}, \mathsf{range}})$

Moreover, from axioms (2.24), (2.25), (2.47) and (2.48), we obtain four FKs:

$\mathsf{Ale} \sqsubseteq \mathsf{Beer} \rightsquigarrow \mathsf{Ale}(\mathsf{id}) \overset{\mathsf{fk}}{\Rightarrow} \mathsf{Beer}(\mathsf{id})$

$\mathsf{Lager} \sqsubseteq \mathsf{Beer} \rightsquigarrow \mathsf{Lager}(\mathsf{id}) \overset{\mathsf{fk}}{\Rightarrow} \mathsf{Beer}(\mathsf{id})$

$\mathsf{hasColour} \sqsubseteq \mathsf{hasDescription} \rightsquigarrow \mathsf{hasColour}(\mathsf{domain}, \mathsf{range}) \overset{\mathsf{fk}}{\Rightarrow} \mathsf{hasDescription}(\mathsf{domain}, \mathsf{range})$

$\mathsf{hasFlavour} \sqsubseteq \mathsf{hasDescription} \rightsquigarrow \mathsf{hasFlavour}(\mathsf{domain}, \mathsf{range}) \overset{\mathsf{fk}}{\Rightarrow} \mathsf{hasDescription}(\mathsf{domain}, \mathsf{range})$

4.5 Generating Logical Triggers

Now, the ATIDB is with the canonical schema established, and we are ready to generate triggers from axioms in a classified T-Box. As we have already illustrated in Section 4.3, triggers can be invoked either before or after an insert to the ATIDB, so that type inference is automatically performed. We denote triggers logically by ECA rules, and we demonstrate the transformation process from OWL to logical triggers as:

$$\mathsf{OWL} \rightsquigarrow \textbf{when } \langle \mathsf{event} \rangle \textbf{ if } \langle \mathsf{condition} \rangle \textbf{ then } \langle \mathsf{action} \rangle$$

where $\langle \mathsf{event} \rangle$ in this chapter denotes the insert of an A-Box fact[1] to a table representing a particular class or property. We introduce two types of events: if $\langle \mathsf{event} \rangle$ begins with the symbol "$-$", then $\langle \mathsf{condition} \rangle$ and $\langle \mathsf{action} \rangle$ are executed before the insert; while for the $\langle \mathsf{event} \rangle$ prefixed with a "$+$" symbol, $\langle \mathsf{condition} \rangle$ and $\langle \mathsf{action} \rangle$ are executed after the insert. We call the former **before triggers**, and the latter **after triggers**. The field of $\langle \mathsf{condition} \rangle$ contains Datalog-style queries, and only if they are evaluated as true, should the $\langle \mathsf{action} \rangle$, which is one of insert, ignore and $\mathsf{rollback}$, be executed.

[1] As we illustrate in Chapter 5, for handling deletes, $\langle \mathsf{event} \rangle$ and $\langle \mathsf{action} \rangle$ can also denote a delete of an A-Box fact or an update of data state.

4.5.1 OWL Classes and Properties

Due to the open-world nature of inference, the same fact might be repeatedly inferred; thus, the most basic trigger we create for each class or property table is to ignore duplicated inserts (recall that from the canonical schema, every class C and every property P result as a table $C(\underline{id})$ and a table $P(\underline{domain, range})$, respectively). These triggers are created based on the following transformation rules:

Class: $C \rightsquigarrow$ **when** $^{-}C(x)$ **if** $\neg C(x)$ **then** $C(x)$ **else** ignore

Property: $P \rightsquigarrow$ **when** $^{-}P(x, y)$ **if** $\neg P(x, y)$ **then** $P(x, y)$ **else** ignore

Based on the two rules, a before trigger is created for each class or property table, and it is invoked before an insert is actually done. The before trigger checks the existence of the data which users attempt to insert; it will allow the insert if the data is not present in the table; otherwise, it will ignore the insert. Note that in logical triggers, free variables in the field of ⟨condition⟩, such as x in **if** $\neg C(x)$ or x and y in **if** $\neg P(x, y)$, are universally quantified.

4.5.2 The Semantics of Class Axioms

We list in this section how logical triggers are generated from class-related axioms (e.g. SubClassOf, EquivalentClasses and DisjointWith axioms). As our approach restricts itself to focus on the OWL 2 RL profile, we ignore the analysis for some semantics which are not allowed in this profile, although we have reviewed them in Section 2.3.2 of Chapter 2.

SubClassOf and EquivalentClasses:

Class-related axioms are formed by specifying a subsumption relation (i.e. SubClassOf) between two class expressions as $C_{E_1} \sqsubseteq C_{E_2}$. For the moment, we only discuss the special case of using SubClassOf to relate two atomic classes (i.e. $C \sqsubseteq D$). The transformations from the more general case of $C_{E_1} \sqsubseteq C_{E_2}$ (e.g. an AllValuesFrom axiom $C \sqsubseteq \forall P.C_E$) to triggers are eventually demonstrated later in this section. The transformation rule for handling $C \sqsubseteq D$ is:

SubClassOf: $C \sqsubseteq D \rightsquigarrow$ **when** $^{+}C(x)$ **if** true **then** $D(x)$

As can be seen, the "+" symbol in the field of ⟨event⟩ identifies this trigger as an after trigger, which is invoked after x has been inserted to the subclass table C, and an attempt at inserting x to the super-class table D is generated. We set the ⟨condition⟩ to true (the trigger can be simplified as **when** $^{+}C(x)$ **then** $D(x)$), because the attempt at inserting $D(x)$ will be checked anyway by the before trigger created on D. Take (2.24) as an example, this axiom expresses a subsumption relation from Ale to Beer, and one after trigger should be generated:

$$\mathsf{Ale} \sqsubseteq \mathsf{Beer} \rightsquigarrow \textbf{when } {}^{+}\mathsf{Ale}(x) \textbf{ then } \mathsf{Beer}(x)$$

For the constructor EquivalentClasses, we also only discuss the case of using it to connect two atomic classes $C \equiv D$, rather than the general case of relating two class expressions $C_{E_1} \equiv C_{E_2}$. Because the equivalence relation can be classified as two subsumption relations, $C \equiv D$ is translated as two logical after triggers:

$$\begin{aligned}\mathsf{EquivalentClasses}\text{: } C \equiv D \rightsquigarrow \textbf{when } {}^{+}C(x) \textbf{ then } D(x)\\ \textbf{when } {}^{+}D(x) \textbf{ then } C(x)\end{aligned}$$

Thus, after a data item has been inserted into one of the tables which represent the two equivalent classes, triggers will attempt an insert of the same data to the other class table. For example, the axiom (2.28), which identifies that Beer and LiquidBread are EquivalentClasses, results in two after triggers:

$$\begin{aligned}\mathsf{Beer} \equiv \mathsf{LiquidBread} \rightsquigarrow \textbf{when } {}^{+}\mathsf{Beer}(x) \textbf{ then } \mathsf{LiquidBread}(x)\\ \textbf{when } {}^{+}\mathsf{LiquidBread}(x) \textbf{ then } \mathsf{Beer}(x)\end{aligned}$$

DisjointClasses, UnionOf and DisjointUnion:

OWL 2 RL allows an ontology user to express the fact that several classes do not contain the same individuals (i.e. these classes are DisjointClasses denoted as $\mathsf{DisCla}(C_1, \ldots, C_n)$). As we have introduced in Chapter 3, if any two classes from $C_1, \ldots, C_n$ contain the same individuals, the ontology will result in an inconsistent state by a tableaux-based reasoner. Therefore, when transforming these semantics into triggers, the trigger created on each table representing each of $C_1, \ldots, C_n$ will check before an insert to determine whether the insert of data is already present in other tables, if so the insert will be rolled back. The transformation rule is:

$$\text{DisjointClasses: } \mathsf{DisCla}(C_1, \ldots, C_n) \rightsquigarrow \textbf{when } {}^{-}C_i(x)\ 1 \leq i \leq n \textbf{ if } C_1(x) \text{ or } \ldots \text{ or } C_{i-1}(x) \text{ or } C_{i+1}(x) \text{ or } \ldots \text{ or } C_n(x) \textbf{ then rollback}$$

Note that there is a difference for handling $\mathsf{DisCla}(C_1, \ldots, C_n)$ between a tableaux-based reasoner and SQOWL2. A tableaux-based reasoner will raise an inconsistent warning if any two classes C_i and C_j (where $i \neq j$) in the disjoint class list $C_1, \ldots, C_n$ contain the same individual x. However, SQOWL2 prevents the ATIDB entering an inconsistent state by rolling back database transactions which attempt to store x in both C_i and C_j simultaneously: 1) if two separate database transactions attempt to insert x into C_i first and then C_j (or into C_j before C_i), respectively, the first transaction will be allowed but the second one will be rolled back. In this case, the ATIDB might end differently if we load the facts as separate transactions in different orders; 2) if the two inserts are gathered as one database transaction, the whole transaction will be rolled back. We argue this difference between SQOWL2 and a tableaux-based reasoner is just operational but not fundamental.

In the beer ontology, we have the axiom (2.29) to specify Ale and Lager are disjoint from each other, and the transformation rule which deals with this axiom is:

$$\mathsf{DisCla}(\mathsf{Ale}, \mathsf{Lager}) \rightsquigarrow \textbf{when } {}^{-}\mathsf{Ale}(x) \textbf{ if } \mathsf{Lager}(x) \textbf{ then rollback}$$
$$\textbf{when } {}^{-}\mathsf{Lager}(x) \textbf{ if } \mathsf{Ale}(x) \textbf{ then rollback}$$

With regard to DisjointUnion, an axiom formed by this constructor actually denotes an equivalent relation between an atomic class and a union of several disjoint classes. This constructor is not allowed in OWL 2 RL, as this profile only allows the UnionOf expression to be a subclass expression but not a super-class expression. For example, $\mathsf{DisUni}(C, D_1, D_2)$ expresses a class C is the DisjointUnion of D_1 and D_2, which is the simpler form of the following two DL statements:

$$D_1 \sqcap D_2 \sqsubseteq \bot \qquad\qquad C \equiv D_1 \sqcup D_2$$

where the left-hand statement expresses D_1 and D_2 are disjoint from each other, and the right-hand statement expresses the equivalence between C and the UnionOf D_1 and D_2 (which can be alternatively expressed as two axioms $D_1 \sqcup D_2 \sqsubseteq C$ and $C \sqsubseteq D_1 \sqcup D_2$). However, OWL 2 RL

does not allow $D_1 \sqcup D_2$ as a super-class expression (i.e. $C \sqsubseteq D_1 \sqcup D_2$ is not allowed), because this brings non-deterministic inference (i.e. for a fact of x in C, we cannot determine which one of D_1 and D_2 or even both of them the fact x belongs to). Thus, if an ontology beyond OWL 2 RL contains $\mathsf{DisUni}(C, D_1, D_2)$, triggers generated by the transformation rule below are unable to capture the semantics of $C \sqsubseteq D_1 \sqcup D_2$.

$$\begin{aligned} \mathsf{DisjointUnion}\colon \mathsf{DisUni}(C, D_1, D_2) \rightsquigarrow\ & \textbf{when } {}^{-}D_1(x) \textbf{ if } D_2(x) \textbf{ then } \mathsf{rollback} \\ & \textbf{when } {}^{-}D_2(x) \textbf{ if } D_1(x) \textbf{ then } \mathsf{rollback} \\ & \textbf{when } {}^{+}D_1(x) \textbf{ then } C(x) \\ & \textbf{when } {}^{+}D_2(x) \textbf{ then } C(x) \end{aligned}$$

The first two triggers capture the semantics of $D_1 \sqcap D_2 \sqsubseteq \bot$ (i.e. DisjointClasses), and the last two triggers are translated from the semantics $D_1 \sqcup D_2 \sqsubseteq C$, which is equivalent to $D_1 \sqsubseteq C$ and $D_2 \sqsubseteq C$. In the beer ontology, the axiom (2.32), which defines Beer as the DisjointUnion of Ale and Lager, will be translated into four triggers as follows:

$$\begin{aligned} \mathsf{DisUni}(\mathsf{Beer}, \mathsf{Ale}, \mathsf{Lager}) \rightsquigarrow\ & \textbf{when } {}^{-}\mathsf{Ale}(x) \textbf{ if } \mathsf{Lager}(x) \textbf{ then } \mathsf{rollback} \\ & \textbf{when } {}^{-}\mathsf{Lager}(x) \textbf{ if } \mathsf{Ale}(x) \textbf{ then } \mathsf{rollback} \\ & \textbf{when } {}^{+}\mathsf{Ale}(x) \textbf{ then } \mathsf{Beer}(x) \\ & \textbf{when } {}^{+}\mathsf{Lager}(x) \textbf{ then } \mathsf{Beer}(x) \end{aligned}$$

As can be seen, the two before triggers are exactly the same as those ones transformed from $\mathsf{DisCla}(\mathsf{Ale}, \mathsf{Lager})$ shown before, and the two after triggers are the same as the triggers created from (2.24) and (2.25), respectively.

AllValuesFrom, SomeValuesFrom, HasValue and OneOf:

OWL 2 RL allows expressions constructed by AllValuesFrom (i.e. $\forall P.C_E$), SomeValuesFrom (i.e. $\exists P.C_E$), HasValue (i.e. $\exists P.\{a\}$); however, it restricts the positions that these expressions can appear in a subsumption relation. For example, OWL 2 RL only allows an AllValuesFrom expression $\forall P.C_E$ to be a super class (i.e. $C \sqsubseteq \forall P.C_E$), but not a subclass (i.e. $\forall P.C_E \sqsubseteq C$). The semantics of $C \sqsubseteq \forall P.C_E$ specifies that if an instance x of C appears in $\langle x, y \rangle$ of P, then y must be inferred as a member of C_E. Thus, the transformation rule for $C \sqsubseteq \forall P.C_E$ is:

$$\text{AllValuesFrom: } C \sqsubseteq \forall P.C_E \rightsquigarrow \textbf{when } {}^{+}C(x) \textbf{ if } P(x,y) \textbf{ then } C_E(y)$$
$$\textbf{when } {}^{+}P(x,y) \textbf{ if } C(x) \textbf{ then } C_E(y)$$

The trigger on C guarantees that after x has been inserted into C, the trigger will attempt an insert of y into C_E if $\langle x, y\rangle$ is present in the table P; the other trigger on P will attempt an insert of y into C_E after inserting $\langle x, y\rangle$ into P if x is already in C.

For the case of $\forall P.C_E \sqsubseteq C$, it is interesting to mention that even if this case was permitted in OWL 2 RL, we actually could not perform any type inference from it. The reason is that, due to the OWA, even if $\forall P.C_E \sqsubseteq C$ holds for a set of x (i.e. for all existing $\langle x, y\rangle$ in P, y is in C_E), there might still exist some not yet recorded $\langle x, z\rangle$ in which z is not a member of C_E. Therefore, the transformation rule for $\forall P.C_E \sqsubseteq C$ generates no triggers:

$$\text{AllValuesFrom: } \forall P.C_E \sqsubseteq C \rightsquigarrow -$$

With regard to a SomeValuesFrom expression $\exists P.C_E$, OWL 2 RL restricts the expression as only a subclass expression (i.e. $\exists P.C_E \sqsubseteq C$) but not a super-class expression (i.e. $C \sqsubseteq \exists P.C_E$). The semantics of $\exists P.C_E \sqsubseteq C$ means that if there exists some $\langle x, y\rangle$ with y as an instance of C_E, then x should be inferred as a member of C; thus, the transformation rule is specified as:

$$\text{SomeValuesFrom: } \exists P.C_E \sqsubseteq C \rightsquigarrow \textbf{when } {}^{+}P(x,y) \textbf{ if } C_E(y) \textbf{ then } C(x)$$
$$\textbf{when } {}^{+}C_E(y) \textbf{ if } P(x,y) \textbf{ then } C(x)$$

Thus, after $\langle x, y\rangle$ has been inserted into P, the trigger on P checks for the presence of y in C_E, and if so, it will insert x into C. The other trigger on C_E checks for the existence of $\langle x, y\rangle$ in P after inserting y into C_E, and if so it will attempt an insert of x into C.

The semantics of $C \sqsubseteq \exists P.C_E$ states that for every x in C, the property P must contain at least one tuple $\langle x, y\rangle$ in which y comes from C_E. However, this might result in inference of anonymous individuals (i.e. non-deterministic inference). To explain this, suppose we have an instance x in C, and there is no tuple $\langle x, y\rangle$ recorded in P. Based on the semantics of $C \sqsubseteq \exists P.C_E$, we only know there must be at least one tuple $\langle x, y\rangle$ (y is an instance of C_E) which is not yet recorded in P. However, we cannot determine which individual in C_E the y refers to; therefore, the transformation rule generates no triggers from $C \sqsubseteq \exists P.C_E$:

$$\text{SomeValuesFrom: } C \sqsubseteq \exists P.C_E \rightsquigarrow -$$

Alternatively, if there is no $\langle x, y\rangle$ recorded in P (or there is $\langle x, y\rangle$ in P but without recording y as a member of C_E), we can insert a tuple $\langle x, \_\rangle$ after such a data item x has been inserted into C, in order to preserve the existential semantics of $C \sqsubseteq \exists P.C_E$. This will provide more complete answers to queries asking for the subjects of P. However, if later such $\langle x, y\rangle$ with y from C_E is recorded in P, the tuple $\langle x, \_\rangle$ should be removed. This alternative way results in triggers:

$$\begin{aligned}\text{SomeValuesFrom: } C \sqsubseteq \exists P.C_E \rightsquigarrow\ & \textbf{when } {}^{+}C(x) \textbf{ if } \neg P(x, y) \text{ or } P(x, y), \neg C_E(y) \textbf{ then } P(x, \_)\\ & \textbf{when } {}^{-}P(x, y) \textbf{ if } P(x, \_), C_E(y) \textbf{ then } \neg P(x, \_), P(x, y)\\ & \textbf{when } {}^{-}C_E(y) \textbf{ if } P(x, \_), P(x, y) \textbf{ then } \neg P(x, \_)\end{aligned}$$

As can be seen, the after trigger on C is used for inserting $\langle x, \_\rangle$ into P, and two before triggers on P and C_E are used for checking if $P(x, \_)$ should be removed. Note that for the case of $C \sqsubseteq \exists P$, since the SomeValuesFrom expression is unqualified, we will remove $P(x, \_)$ as soon as a tuple like $\langle \mathsf{x}, \mathsf{y}\rangle$ is inserted into P; thus the transformation rule should be modified as:

$$\begin{aligned}\text{SomeValuesFrom: } C \sqsubseteq \exists P \rightsquigarrow\ & \textbf{when } {}^{+}C(x) \textbf{ if } \neg P(x, y) \textbf{ then } P(x, \_)\\ & \textbf{when } {}^{-}P(x, y) \textbf{ if } P(x, \_) \textbf{ then } P(x, y)\end{aligned}$$

As a special case of SomeValueFrom, a HasValue expression $\exists P.\{a\}$ replaces the C_E in $\exists P.C_E$ by a class containing only one individual $\{a\}$. In other words, C_E is defined by enumerating its contained individuals by using the constructor OneOf. Since the individual a is the only member of this anonymous class, and it is explicitly defined, there is no restriction in OWL 2 RL in terms of using $\exists P.\{a\}$ i.e. both $C \sqsubseteq \exists P.\{a\}$ and $\exists P.\{a\} \sqsubseteq C$ are allowed, and the transformation rules for dealing with both are:

$$\begin{aligned}\text{HasValue: } C \sqsubseteq \exists P.\{a\} \rightsquigarrow\ & \textbf{when } {}^{+}C(x) \textbf{ then } P(x, a)\\ \exists P.\{a\} \sqsubseteq C \rightsquigarrow\ & \textbf{when } {}^{+}P(x, a) \textbf{ then } C(x)\end{aligned}$$

The after trigger on C attempts an insert of $\langle x, a\rangle$ into P after x has been inserted into C; the after trigger on P will attempt an insert of x into C after $\langle x, a\rangle$ has been inserted into P. For example, in the beer ontology, we have an axiom (2.37), which denotes Porter as a subclass of $\exists$hasColour.{Dark}, and the transformation rule for this is:

$$\mathsf{Porter} \sqsubseteq \exists \mathsf{hasColour}.\{\mathsf{Dark}\} \rightsquigarrow \textbf{when } {}^{+}\mathsf{Porter}(x) \textbf{ then } \mathsf{hasColour}(x, \text{"Dark"})$$

As we have just mentioned, the constructor OneOf can be used to define an anonymous class by enumerating all of its included individuals (expressed as $\{a_1 \dots a_n\}$). For such an anonymous class, we create a table named by concatenating all of its individuals' names, and also store exactly the individuals defined by OneOf. Then, we create a before trigger to prevent inserts which attempt to add other individuals not in the enumeration (where C_{ac} denotes the table name for the anonymous class):

$$\mathsf{OneOf}\colon \{a_1 \dots a_n\} \rightsquigarrow \textbf{when } {}^{-}C_{ac}(x) \textbf{ if } \neg C_{ac}(x) \textbf{ then } \mathsf{rollback}$$

Cardinality**:**

OWL 2 RL has a restriction on using cardinality-related constructors. In essence, it only supports the use of $C \sqsubseteq \leqslant n\, P$ and $C \sqsubseteq \leqslant n\, P.C_E$ where $n = 0/1$. When $n = 0$ in the unqualified case of $C \sqsubseteq \leqslant n\, P$, any individual x from the class C must not appear in any tuple of $\langle x, y \rangle$ in P; thus, the transformation rule is:

$$\begin{aligned}\mathsf{MaxCardinality}\colon C \sqsubseteq \leqslant 0\, P \rightsquigarrow \textbf{when } {}^{-}C(x) \textbf{ if } P(x,y) \textbf{ then } \mathsf{rollback}\\ \textbf{when } {}^{-}P(x,y) \textbf{ if } C(x) \textbf{ then } \mathsf{rollback}\end{aligned}$$

The first trigger checks whether $\langle x, y \rangle$ is already in P before inserting x into C, if so the transaction will be rolled back. The second trigger will roll back an insert of $\langle x, y \rangle$ to P if x has been stored in C. These two triggers together prevent the ATIDB from entering an inconsistent state. For the qualified case $C \sqsubseteq \leqslant 0\, P.C_E$, it is slightly more complicated, as in order to keep the ontology consistent, we should ensure for each x in C, there is no such $\langle x, y \rangle$ in P where y is in C_E. This causes the transformation rule to generate three triggers:

$$\begin{aligned}\mathsf{MaxCardinality}\colon C \sqsubseteq \leqslant 0\, P.C_E \rightsquigarrow \textbf{when } {}^{-}C(x) \textbf{ if } P(x,y), C_E(y) \textbf{ then } \mathsf{rollback}\\ \textbf{when } {}^{-}P(x,y) \textbf{ if } C(x), C_E(y) \textbf{ then } \mathsf{rollback}\\ \textbf{when } {}^{-}C_E(y) \textbf{ if } C(x), P(x,y) \textbf{ then } \mathsf{rollback}\end{aligned}$$

The first two triggers are similar to the two triggers created from $C \sqsubseteq \leqslant 0\, P$, but add additional checking of y in C_E in the field of $\langle \mathsf{condition} \rangle$. Because of the non-qualification, the third trigger

will be created on the table C_E as shown above, which rolls back any insert of y to C_E, if $C(x)$ and $P(x, y)$ are true.

With respect to $n = 1$ in the unqualified axiom $C \sqsubseteq \leqslant 1\, P$, we recall its satisfying condition defined by $\mathcal{I}$ of the Direct Semantics:

$$\mathcal{I} \models C \sqsubseteq \leqslant 1\, P \text{ iff } \forall x \in C^{\mathcal{I}}, y \in \Delta, z \in \Delta : \text{if} \langle x, y \rangle \in P^{\mathcal{I}} \text{ and } \langle x, z \rangle \in P^{\mathcal{I}} \text{ then } y = z$$

which results in an inference of a SameIndividual axiom, i.e. for every x in C, if such tuples $\langle x, y \rangle$ and $\langle x, z \rangle$ both exist in P, then we can infer y and z are same individuals. However, since our approach adopts the UNA, which specifies that individuals with different names do not refer to the same object, $C \sqsubseteq \leqslant 1\, P$ will be translated to two before triggers on tables C and P:

MaxCardinality: $C \sqsubseteq \leqslant 1\, P \rightsquigarrow$ **when** $^{-}C(x)$ **if** count$[P(x, _)] > 1$ **then** rollback

when $^{-}P(x, y)$ **if** $C(x), P(x, z), z \neq y$ **then** rollback

These triggers essentially roll back any insert which will make the ATIDB violate the UNA. The trigger on C will roll back an insert of x, if x is already related to at least two different individuals (i.e. count$[P(x, _)] > 1$). The trigger on P will not allow an insert of $\langle x, y \rangle$, if x is in C and is already related to some other z by P.

The qualified case is expressed as $C \sqsubseteq \leqslant 1\, P.C_E$, and the satisfying condition is:

$$\mathcal{I} \models C \sqsubseteq \leqslant 1\, P.C_E \text{ iff } \forall x \in C^{\mathcal{I}}, y \in {C_E}^{\mathcal{I}}, z \in {C_E}^{\mathcal{I}} : \text{if} \langle x, y \rangle \in P^{\mathcal{I}} \text{ and } \langle x, z \rangle \in P^{\mathcal{I}} \text{ then } y = z$$

Hence, in order to obey the UNA, we need to ensure that P relates any individual x from C to at most one individual from C_E. Therefore, the transformation rule for $C \sqsubseteq \leqslant 1\, P.C_E$ will generate three triggers on each of C, P and C_E:

MaxCardinality: $C \sqsubseteq \leqslant 1\, P.C_E \rightsquigarrow$ **when** $^{-}C(x)$ **if** count$[P(x, y), C_E(y)] > 1$ **then** rollback

when $^{-}P(x, y)$ **if** $C(x), P(x, z), C_E(y), C_E(z), y \neq z$ **then** rollback

when $^{-}C_E(z)$ **if** $C(x), P(x, y), P(x, z), C_E(y), y \neq z$ **then** rollback

IntersectionOf:

OWL 2 RL allows the specifying of a class as equivalent to an IntersectionOf expression between several class expressions, which can be either atomic or complex, but should obey the usage

restrictions defined in the language profile. The general DL syntax for this is $C \equiv C_{E_1} \sqcap \ldots \sqcap C_{E_n}$; for example, axiom (1.1) defines CzechLager as the IntersectionOf classes Lager and CzechBeer.

By applying a tableaux-based reasoner, the statement $C \equiv C_{E_1} \sqcap \ldots \sqcap C_{E_n}$ is classified to $C \sqsubseteq C_{E_1} \sqcap \ldots \sqcap C_{E_n}$ and $C_{E_1} \sqcap \ldots \sqcap C_{E_n} \sqsubseteq C$. The former can be simply interpreted as n SubClassOf axioms $C \sqsubseteq C_{E_i}$ where $1 \leq i \leq n$:

$$C \sqsubseteq C_{E_1} \sqcap \ldots \sqcap C_{E_n} \rightsquigarrow C \sqsubseteq C_{E_i} \text{ where } 1 \leq i \leq n$$

Translating each of $C \sqsubseteq C_{E_i}$ $(1 \leq i \leq n)$ into triggers has been already demonstrated previously. However, the subsumption relation from an IntersectionOf expression $C_{E_1} \sqcap \ldots \sqcap C_{E_n}$ to a class C (e.g. the axiom (1.7)) is more complex to translate into triggers, as after executing each insert of x into one of C_{E_i} $(1 \leq i \leq n)$, triggers should check whether all other C_{E_j} $(j \neq i)$ holds for x, and if so, the x should be inserted into C. We specify the transformation rule as:

IntersectionOf: $C_{E_1} \sqcap \ldots \sqcap C_{E_n} \sqsubseteq C \rightsquigarrow$

$$\textbf{when } \mathsf{trigger}(C_{E_i})\ 1 \leq i \leq n \textbf{ if } \mathsf{holds}(C_{E_1}), \ldots, \mathsf{holds}(C_{E_n}) \textbf{ then } C(x)$$

where the trigger() function specifies the tables to trigger on, and the holds() function maps each component C_{E_i} $(1 \leq i \leq n)$ of the IntersectionOf expression into predicate logic:

$\mathsf{trigger}(C) := {}^{+}C(x)$ $\qquad$ $\mathsf{holds}(C) := C(x)$

$\mathsf{trigger}(\exists P.\{a\}) := {}^{+}P(x, a)$ $\qquad$ $\mathsf{holds}(\exists P.\{a\}) := P(x, a)$

$\mathsf{trigger}(\exists P.C_E) := {}^{+}P(x, y)$ and ${}^{+}C_E(y)$ $\qquad$ $\mathsf{holds}(\exists P.C_E) := P(x, y), C_E(y)$

Note that we omit certain cases that C_{E_i} $(1 \leq i \leq n)$ are expressions which are not allowed in OWL 2 RL, such as AllValueFrom expressions $\forall P.C_E$. As an example of using the above transformation rule, axiom (1.7) (i.e. Lager $\sqcap$ CzechBeer $\sqsubseteq$ CzechLager) will be expanded out as:

$$\textbf{when } \mathsf{trigger}(\mathsf{Lager}) \textbf{ if } \mathsf{holds}(\mathsf{Lager}), \mathsf{holds}(\mathsf{CzechBeer}) \textbf{ then } \mathsf{CzechLager}(x)$$

$$\textbf{when } \mathsf{trigger}(\mathsf{CzechBeer}) \textbf{ if } \mathsf{holds}(\mathsf{Lager}), \mathsf{holds}(\mathsf{CzechBeer}) \textbf{ then } \mathsf{CzechLager}(x)$$

and continuously expanding the trigger() and holds() functions gives:

$$\textbf{when } {}^{+}\mathsf{Lager}(x) \textbf{ if } \mathsf{Lager}(x), \mathsf{CzechBeer}(x) \textbf{ then } \mathsf{CzechLager}(x)$$

$$\textbf{when } {}^{+}\mathsf{CzechBeer}(x) \textbf{ if } \mathsf{Lager}(x), \mathsf{CzechBeer}(x) \textbf{ then } \mathsf{CzechLager}(x)$$

If we remove the redundant check on Lager and CzechBeer in the ⟨condition⟩ of the first and sec-

ond triggers, respectively, we will obtain the two triggers we have illustrated for the axiom (1.7) in Section 4.3.2.

4.5.3 The Semantics of Property Axioms

In this section, we address the transformations rules which handle property-related axioms. As we have reviewed in Chapter 2, some of these axioms are actually relations between sets of individuals (e.g. PropertyDomain, PropertyRange, *etc.*), and not between sets of tuples.

InverseOf and InverseProperty:

Unlike complex class expressions, which can be formed by numerous constructors provided by OWL, complex property expressions can be only InverseOf expressions, which are denoted as P^- in DL. P^- specifies a set of tuples which swap the subjects and objects in P. A very common usage of InverseOf is to make an atomic property P equivalent to the InverseOf another property Q (i.e. $P \equiv Q^-$). Such a usage defines P as an InverseProperty of Q and vice versa. Handling $P \equiv Q^-$ (two properties are inverse of each other) by SQOWL2 is achieved by the following transformation rule:

$$\begin{aligned} \text{InverseProperty: } P \equiv Q^- \rightsquigarrow\ & \textbf{when } {}^+P(x,y) \textbf{ then } Q(y,x) \\ & \textbf{when } {}^+Q(x,y) \textbf{ then } P(y,x) \end{aligned}$$

After $\langle x, y\rangle$ has been inserted into one of the property tables, the trigger will attempt an insert of $\langle y, x\rangle$ into the other property table. For example, SQOWL2 generates the two triggers below from (2.40), which defines isFermentedBy and fermentsBeer are InverseOf each other:

$$\begin{aligned} \text{isFermentedBy} \equiv \text{fermentsBeer}^- \rightsquigarrow\ & \textbf{when } {}^+\text{isFermentedBy}(x,y) \textbf{ then } \text{fermentsBeer}(y,x) \\ & \textbf{when } {}^+\text{fermentsBeer}(x,y) \textbf{ then } \text{isFermentedBy}(y,x) \end{aligned}$$

Note that the before triggers created on all property tables prevent the insertion of duplicated tuples, so that the two after triggers created from $P \equiv Q^-$ will not cause an infinite loop.

PropertyDomain **and** PropertyRange**:**

Apart from using InverseOf to specify two inverse properties, expressions formed by this constructor can be used to express other types of axioms. PropertyDomain axioms $\top \sqsubseteq \forall P^{-}.C_{E_1}$ use the InverseOf expression P^{-} to restrict the subject values of a property P to individuals of a particular class C_{E_1}. Based on the semantics, an after trigger is generated on P to trigger an insert of x to C_{E_1} after $\langle x, y\rangle$ has been inserted into P. A PropertyRange $\top \sqsubseteq \forall P.C_{E_2}$, by contrast, restricts the object values of P to a particular class C_{E_2}. Another trigger should be created on P to insert y into C_{E_2} after $\langle x, y\rangle$ has been stored in P. Such a transformation process is shown as:

$$\text{PropertyDomain: } \top \sqsubseteq \forall P^{-}.C_{E_1} \rightsquigarrow \textbf{when } {}^{+}P(x, y) \textbf{ then } C_{E_1}(x)$$

$$\text{PropertyRange: } \top \sqsubseteq \forall P.C_{E_2} \rightsquigarrow \textbf{when } {}^{+}P(x, y) \textbf{ then } C_{E_2}(y)$$

Recalling the AllValuesFrom axiom $C \sqsubseteq \forall P.C_E$, which has been analysed when C is an atomic class (i.e. not $\top$), both PropertyDomain and PropertyRange axioms can be treated as special cases of $C \sqsubseteq \forall P.C_E$ where C is set as $\top$. Thus x and y in the above two triggers are all universally quantified.

For example, the axioms (2.41) and (2.42) respectively specifying the PropertyDomain and PropertyRange of the property isFermentedBy are translated to an after trigger:

$$\left.\begin{array}{l}\top \sqsubseteq \forall \mathsf{isFermentedBy}^{-}.\mathsf{Beer} \\ \top \sqsubseteq \forall \mathsf{isFermentedBy}.\mathsf{Yeast}\end{array}\right\} \rightsquigarrow \textbf{when } {}^{+}\mathsf{isFermentedBy}(x, y) \textbf{ then } \mathsf{Beer}(x), \mathsf{Yeast}(y)$$

As can be seen from this transformation rule, we merge two logical triggers into one, as the attempt at inserting both x into Beer and y into Yeast are after persisting $\langle x, y\rangle$ in the same table isFermentedBy. Note that as we will show in Chapter 7, if multiple after triggers are required after inserting into the same table, we merge them together as one single after trigger. A similar merging rule also applies when generating before triggers. This leads to a more efficient implementation of the physical triggers.

SubPropertyOf, EquivalentProperties and DisjointProperties:

To express relations of subsumption, equivalence and disjointness between property expressions (can be atomic or complex), we use constructors SubPropertyOf, EquivalentProperties and DisjointProperties (analogous to SubClassOf, EquivalentClasses and DisjointClasses), respectively. Here, we only discuss the case of atomic properties, axioms involving complex properties are illustrated later when explaining specific constructors.

The DL syntax $P \sqsubseteq Q$ shows that the property P is a SubPropertyOf another property Q, which specifies that instances contained in P must be also in Q. $P \sqsubseteq Q$ results in an after trigger on P, which attempts an insert of $\langle x, y \rangle$ into Q, after $\langle x, y \rangle$ has been inserted into P:

$$\text{SubPropertyOf: } P \sqsubseteq Q \rightsquigarrow \textbf{when } {}^{+}P(x, y) \textbf{ then } Q(x, y)$$

For example, the two SubPropertyOf axioms (2.47) and (2.48) in the beer ontology are translated into two after triggers:

$$\mathsf{hasColour} \sqsubseteq \mathsf{hasDescription} \rightsquigarrow \textbf{when } {}^{+}\mathsf{hasColour}(x, y) \textbf{ then } \mathsf{hasDescription}(x, y)$$

$$\mathsf{hasFlavour} \sqsubseteq \mathsf{hasDescription} \rightsquigarrow \textbf{when } {}^{+}\mathsf{hasFlavour}(x, y) \textbf{ then } \mathsf{hasDescription}(x, y)$$

Due to the fact that an equivalence relation can be treated as two subsumption relations, translating an EquivalentProperties axiom $P \equiv Q$ (specified below) is equivalent to translating two SubPropertyOf axioms $P \sqsubseteq Q$ and $Q \sqsubseteq P$.

$$\begin{aligned}\text{EquivalentProperties: } P \equiv Q \rightsquigarrow \textbf{when } {}^{+}P(x, y) \textbf{ then } Q(x, y)\\ \textbf{when } {}^{+}Q(x, y) \textbf{ then } P(x, y)\end{aligned}$$

The two after triggers ensure that tables P and Q store exactly the same instances. For example, in the beer ontology, the EquivalentProperties axiom (2.49) is mapped into two after triggers:

$$\begin{aligned}\mathsf{hasFlavour} \equiv \mathsf{hasTaste} \rightsquigarrow \textbf{when } {}^{+}\mathsf{hasFlavour}(x, y) \textbf{ then } \mathsf{hasTaste}(x, y)\\ \textbf{when } {}^{+}\mathsf{hasTaste}(x, y) \textbf{ then } \mathsf{hasFlavour}(x, y)\end{aligned}$$

Finally, the disjointness is represented by using the constructor DisjointProperties, and DL provides a syntax of $\text{DisPro}(P_1, \ldots, P_n)$ to denote properties $P_1, \ldots, P_n$ are disjoint from each other. The semantics of disjointness is achieved by before triggers, which reject inserts of the same $\langle x, y \rangle$ to one of the properties, if $\langle x, y \rangle$ has already been inserted to one of the other

disjoint properties. Translating DisPro($P_1, \ldots, P_n$) into before triggers is defined as:

DisjointProperties: **when** $^{-}P_i(x, y)$ $1 \leq i \leq n$

DisPro($P_1, \ldots, P_n$) $\rightsquigarrow$ **if** $P_1(x, y)$ or $\ldots$ or $P_{i-1}(x, y)$ or $P_{i+1}(x, y)$ or $\ldots$ or $P_n(x, y)$ **then** rollback

For example, the axiom (2.50) is translated into two before triggers:

DisPro(hasTaste, hasColour) $\rightsquigarrow$ **when** $^{-}$hasTaste(x, y) **if** hasColour(x, y) **then** rollback

when $^{-}$hasColour(x, y) **if** hasTaste(x, y) **then** rollback

Again, similar to DisjointClasses, triggers generated from DisjointProperties prevent the ATIDB entering an inconsistent state. If the same $\langle x, y \rangle$ is inserted into tables $P_1 \ldots P_n$ (representing DisjointProperties $P_1 \ldots P_n$), simultaneously as separate database transactions, only the first transaction of insert will be allowed and the others will be rolled back; if these inserts are gathered as one database transaction, the whole transaction will be rolled back.

PropertyChain and TransitiveProperty:

PropertyChain is a new feature of OWL 2 compared with OWL 1, and it allows a property to be defined as a super property of a concatenation of two or more properties ($P_1 \circ \ldots \circ P_n \sqsubseteq P$). We shall first discuss the simplest case that $P_1 \circ P_2 \sqsubseteq P$ where the chain concatenates two properties. $P_1 \circ P_2$ essentially denotes a set of $\langle x, z \rangle$ such that there exists some $\langle x, y \rangle$ in P_1 and $\langle y, z \rangle$ in P_2. Thus when making $P_1 \circ P_2$ a sub property of P, such tuples $\langle x, z \rangle$ should be inferred as members of P. SQOWL2 translates $P_1 \circ P_2 \sqsubseteq P$ into two after triggers:

PropertyChain: $P_1 \circ P_2 \sqsubseteq P \rightsquigarrow$ **when** $^{+}P_1(x, y)$ **if** $P_2(y, z)$ **then** $P(x, z)$

when $^{+}P_2(y, z)$ **if** $P_1(x, y)$ **then** $P(x, z)$

The trigger on P_1 will check if tuples $\langle y, z \rangle$ exist in P_2 after $\langle x, y \rangle$ has been inserted into P_1, and if so, it will attempt to insert $\langle x, z \rangle$ into P. The other trigger on P_2 will insert $\langle x, z \rangle$ into P, if $\langle x, y \rangle$ is in P_1 after $\langle y, z \rangle$ has been inserted into P_2. In the beer ontology, (2.53) specifying that brewedIn is a super property of the PropertyChain concatenating brewedIn and locatedIn is transformed as:

$$\mathsf{brewedIn} \circ \mathsf{locatedIn} \sqsubseteq \mathsf{brewedIn} \rightsquigarrow \textbf{when } {}^{+}\mathsf{brewedIn}(x,y) \textbf{ if } \mathsf{locatedIn}(y,z) \textbf{ then } \mathsf{brewedIn}(x,z)$$
$$\textbf{when } {}^{+}\mathsf{locatedIn}(y,z) \textbf{ if } \mathsf{brewedIn}(x,y) \textbf{ then } \mathsf{brewedIn}(x,z)$$

A special case of $P_1 \circ P_2 \sqsubseteq P$ is to make the property P a super property of $P \circ P$, which concatenates the property itself, and this P is then called a TransitiveProperty. The axiom $P \circ P \sqsubseteq P$ gives the property the semantics of transitivity, so that if $\langle x, y\rangle$ and $\langle y, z\rangle$ are in P, then $\langle x, z\rangle$ must belong to P. Therefore, a TransitiveProperty P results in two after triggers:

$$\mathsf{TransitiveProperty}\text{: } P \circ P \sqsubseteq P \rightsquigarrow \textbf{when } {}^{+}P(x,y) \textbf{ if } P(y,z) \textbf{ then } P(x,z)$$
$$\textbf{when } {}^{+}P(y,z) \textbf{ if } P(x,y) \textbf{ then } P(x,z)$$

For instance, the transformation rule for the TransitiveProperty locatedIn (i.e. axiom (2.54)) is:

$$\mathsf{locatedIn} \circ \mathsf{locatedIn} \sqsubseteq \mathsf{locatedIn} \rightsquigarrow \textbf{when } {}^{+}\mathsf{locatedIn}(x,y) \textbf{ if } \mathsf{locatedIn}(y,z) \textbf{ then } \mathsf{locatedIn}(x,z)$$
$$\textbf{when } {}^{+}\mathsf{locatedIn}(y,z) \textbf{ if } \mathsf{locatedIn}(x,y) \textbf{ then } \mathsf{locatedIn}(x,z)$$

In general, PropertyChain can be used to concatenate more than two properties (i.e. $P_1 \circ \ldots \circ P_n$), and then a property P can be defined as its super property (i.e. $P_1 \circ \ldots \circ P_n \sqsubseteq P$). Translating such an axiom into triggers is more complex, and is specified as follows:

$$\begin{aligned} P_1 \circ \ldots \circ P_n \sqsubseteq P \rightsquigarrow\ & \textbf{when } {}^{+}P_1(x,y) \textbf{ if } P'_{2,n}(y,z) \textbf{ then } P(x,z) \\ & \textbf{when } {}^{+}P_n(y,z) \textbf{ if } P'_{1,n-1}(x,y) \textbf{ then } P(x,z) \\ & \textbf{when } {}^{+}P_i(p,q)\ 1 < i < n \textbf{ if } P'_{1,i-1}(x,p), P'_{i+1,n}(q,z) \textbf{ then } P(x,z) \\ & \text{where } P'_{m,n}(x,y) = P_m(x,x_1), P_{m+1}(x_1,x_2), \ldots, P_n(x_{n-1},y) \end{aligned}$$

where the first two triggers respectively deal with the case that if there is an insertion to the first and last property tables in the PropertyChain (i.e. P_1 and P_n), the triggers will treat the remaining subchains as a join unit and search data matched from the unit. The third trigger handles the insertion to the middle subchains (i.e. not P_1 or P_n) by creating two join units and then fetching the matching data from them.

SymmetricProperty and AsymmetricProperty:

Although OWL 2 only provides the constructor InverseOf to form complex property expressions, numerous characteristics can be assigned to a property, such as transitivity we have

just introduced. Besides TransitiveProperty, OWL 2 provides several constructors, namely SymmetricProperty, AsymmetricProperty, ReflexiveProperty, IrreflexiveProperty, FunctionalProperty and InverseFunctionalProperty, to characterise a property.

In OWL 2 RL, a property can be symmetric, and such a property is called SymmetricProperty. A property P is defined as symmetric by expressing an equivalence relation between the property and its InverseOf expression (i.e. $P \equiv P^-$). Thus a SymmetricProperty P must contain $\langle y, x \rangle$ if it includes $\langle x, y \rangle$, and translating the semantics of symmetry into triggers is specified as:

$$\text{SymmetricProperty: } P \equiv P^- \rightsquigarrow \textbf{when } {}^+P(x, y) \textbf{ then } P(y, x)$$

An after trigger on P will insert $\langle y, x \rangle$ into P after $\langle x, y \rangle$ has been inserted. Take the symmetric property adjacentPlace (i.e. (2.58)) as an example, the transformation rule is specified as:

$$\text{adjacentPlace} \equiv \text{adjacentPlace}^- \rightsquigarrow \textbf{when } {}^+\text{adjacentPlace}(x, y) \textbf{ then } \text{adjacentPlace}(y, x)$$

Thus, if we load the A-Box fact (2.60) as an insert of $\langle \text{Germany}, \text{Belgium} \rangle$ into adjacentPlace, an insert of $\langle \text{Belgium}, \text{Germany} \rangle$ to adjacentPlace will also be executed, which makes the A-Box fact (2.61) materialised in the ATIDB.

By contrast, an AsymmetricProperty P (i.e. DisPro(P, P^-)) must not contain $\langle y, x \rangle$ if $\langle x, y \rangle$ has already been asserted as a fact of this property. Thus, the transformation rule for DisPro(P, P^-) is specified below, in order to keep the ATIDB consistent:

$$\text{AsymmetricProperty: DisPro}(P, P^-) \rightsquigarrow \textbf{when } {}^-P(x, y) \textbf{ if } P(y, x) \textbf{ then rollback}$$

For instance, the AsymmetricProperty hasDescription specified by (2.62) is translated into a before trigger on the property table hasDescription:

$$\text{DisPro}(\text{hasDescription}, \text{hasDescription}^-) \rightsquigarrow$$
$$\textbf{when } {}^-\text{hasDescription}(x, y) \textbf{ if } \text{hasDescription}(y, x) \textbf{ then rollback}$$

ReflexiveProperty, IrreflexiveProperty and SelfRestriction:

As we have reviewed in Chapter 2, OWL 2 provides a special class called Self to represent the semantics of **reflexivity**, and uses the expression of $\exists P.\text{Self}$ to denote a set of individuals x which are related to themselves by P. OWL 2 allows two types of reflexivity, namely universal

and local reflexivity. A property assigned with universal reflexivity is called a ReflexiveProperty (denoted as $\top \sqsubseteq \exists P.\mathsf{Self}$ in DL), which relates every individual in the ontology to the individual itself. In order to transform the semantics of universal reflexivity, we could create a table Thing(id) for the class Thing to include all individuals in an ontology, and generate a trigger on this table which attempts an insert of $\langle x, x\rangle$ into P after each insertion of x to Thing.

$$\text{ReflexiveProperty: } \top \sqsubseteq \exists P.\mathsf{Self} \rightsquigarrow \textbf{when } {}^{+}\mathsf{Thing}(x) \textbf{ then } P(x, x)$$

This could lead the table Thing to be a very large size, since Thing has to include all individuals in an ontology, and this might make SQOWL2 not scalable when handling large A-Boxes. Indeed, the RL profile does not allow the use of the class Thing nor ReflexiveProperty. Instead, rather than having the extra table Thing, we create for every class table C (not Thing) an after trigger below, which will attempt to insert $\langle x, x\rangle$ into P, after x has been inserted into C:

$$\text{ReflexiveProperty: } \top \sqsubseteq \exists P.\mathsf{Self} \rightsquigarrow \textbf{when } {}^{+}C(x) \textbf{ then } P(x, x)$$

OWL 2 RL does allow the semantics of local reflexivity, which is expressed by a constructor called SelfRestriction. Its DL denotation $C \sqsubseteq \exists P.\mathsf{Self}$ represents the semantics that every individual x from a class C is related to itself by the property P. Thus, to capture this semantic, an after trigger should be created on the table C, and after x has been inserted into C a tuple of $\langle x, x\rangle$ should be inserted into P. We may even specify a subsumption relation from $\exists P.\mathsf{Self}$ to C (i.e. $\exists P.\mathsf{Self} \sqsubseteq C$). This requires another after trigger on P, which attempts to insert x into C after $\langle x, x\rangle$ has been inserted into P. The two triggers are specified as:

$$\text{SelfRestriction: } C \sqsubseteq \exists P.\mathsf{Self} \rightsquigarrow \textbf{when } {}^{+}C(x) \textbf{ then } P(x, x)$$

$$\exists P.\mathsf{Self} \sqsubseteq C \rightsquigarrow \textbf{when } {}^{+}P(x, x) \textbf{ then } C(x)$$

OWL 2 RL allows the use of IrreflexiveProperty to specify that a property does not relate any individual to the same individual. Consequently, we create a before trigger below for an IrreflexiveProperty P (i.e. $\top \sqsubseteq \neg\exists P.\mathsf{Self}$) to roll back any insert of $\langle x, x\rangle$.

$$\text{IreflexiveProperty: } \top \sqsubseteq \neg\exists P.\mathsf{Self} \rightsquigarrow \textbf{when } {}^{-}P(x, y) \textbf{ if } x = y \textbf{ then } \mathsf{rollback}$$

Therefore, the IrreflexiveProperty hasDescription defined by (2.64) will be transformed as follows:

$$\top \sqsubseteq \neg\exists \mathsf{hasDescription}.\mathsf{Self} \rightsquigarrow \textbf{when } {}^{-}\mathsf{hasDescription}(x, y) \textbf{ if } x = y \textbf{ then } \mathsf{rollback}$$

FunctionalProperty, InverseFunctionalProperty and HasKey:

The semantics of a FunctionalProperty implies that if the same individual x is related to two individuals y and z by this property, then y and z are the same individuals (i.e. SameIndividual axioms). To specify that multiple individuals with different names actually refer to the same knowledge object is an important feature of OWL. However, this contradicts the UNA, which states that individuals with different names are different (i.e. which is equivalent to applying DifferentIndividuals over all individuals in OWL). As SQOWL2 follows the UNA (like most database systems), we should reject the inserts which lead to the inference of SameIndividual axioms. Thus, inserting $\langle x, z\rangle$ into P representing a FunctionalProperty should be rolled back if $\langle x, y\rangle$ is already stored in P and $y \neq z$, and this results in a before trigger:

FunctionalProperty: $\top \sqsubseteq \leqslant 1\, P \rightsquigarrow$ **when** $^{-}P(x, z)$ **if** $P(x, y), z \neq y$ **then** rollback

With regard to an InverseFunctionalProperty P, it specifies that the inverse of P is functional (denoted as $\top \sqsubseteq \leqslant 1\, P^{-}$ in DL). In other words, if two individuals y and z are both related to the same individual x by P, then y and z are the same individuals. This again contradicts the UNA, and we handle an InverseFunctionalProperty P by the following trigger:

InverseFunctionalProperty: $\top \sqsubseteq \leqslant 1\, P^{-} \rightsquigarrow$ **when** $^{-}P(z, x)$ **if** $P(y, x), z \neq y$ **then** rollback

HasKey$(C, (P_1 \ldots P_n))$, as another new feature introduced by OWL 2, specifies that multiple properties $P_1 \ldots P_n$ together uniquely identify an individual in a class C (we may call $P_1 \ldots P_n$ key properties of C, and call C the key class of $P_1 \ldots P_n$). Here, to simplify the illustration of triggers, we consider a HasKey axiom specifying two key properties for a class (i.e. HasKey$(C, (P_1\ P_2))$). This implies that if two individuals x and y from C are respectively related to z_1 and z_2 by P_1 and P_2, then x and y are the same individuals. Again, we create the triggers below to roll back any inserts which violate the UNA.

$$\mathsf{HasKey}(C, (P_1\ P_2)) \rightsquigarrow \textbf{when } {}^{-}P_1(x, z_1)$$
$$\textbf{if } C(x), C(y), P_1(y, z_1), P_2(y, z_2), P_2(x, z_2), x \neq y$$
$$\textbf{then } \mathsf{rollback}$$
$$\textbf{when } {}^{-}P_2(x, z_2)$$
$$\textbf{if } C(x), C(y), P_1(y, z_1), P_2(y, z_2), P_1(x, z_1), x \neq y$$
$$\textbf{then } \mathsf{rollback}$$
$$\textbf{when } {}^{-}C(x)$$
$$\textbf{if } C(y), P_1(y, z_1), P_2(y, z_2), P_1(x, z_1), P_2(x, z_2), x \neq y$$
$$\textbf{then } \mathsf{rollback}$$

NegativePropertyAssertion

Finally, OWL 2 RL allows the specification that certain facts $\langle x, y\rangle$ are not in a property P by using the constructor NegativePropertyAssertion. This is denoted as $\neg P(x, y)$ in DL, and should result in a before trigger on the table P to reject the insert of $\langle x, y\rangle$ to this table:

$$\mathsf{NegativePropertyAssertion}\text{: } \neg P(x, y) \rightsquigarrow \textbf{when } {}^{-}P(x, y) \textbf{ then } \mathsf{rollback}$$

4.6 Optimisation

In this section, we further illustrate some optimisations focusing on functional properties, inverse functional properties, and key properties. The optimisations enable SQOWL2 to build a more **conventional schema** in the ATIDB, that is similar to those that would be created via a conventional design process into the **Third Normal Form** (**3NF**) [Bat89] schema, and will allow SQOWL2 to be adapted so that it can be used to generate triggers on top of existing relational schemas (as opposed to generating new schemas with associated triggers).

4.6.1 Canonical Schema to Conventional Schema

SQOWL2 uses the fact that a FunctionalProperty can be stored in the same table which represents the PropertyDomain of the property [BB93], to improve the canonical schema to a more conventional schema. Similarly, we may store all KeyProperties and related triggers to the table representing their KeyClass, and store an InverseFunctionalProperty and related triggers to the table representing its PropertyRange.

Thus after establishing a canonical schema as we have introduced in Section 4.4.1, if the ontology contains properties characterised as a FunctionalProperty or an InverseFunctionalProperty (i.e. the InverseOf expression of a property is a FunctionalProperty) or a KeyProperty (i.e. properties that together can uniquely identify a knowledge object), we alter the canonical schema by 1) storing a FunctionalProperty as an extra column of its PropertyDomain table; 2) storing a InverseFunctionalProperty as an extra column of its PropertyRange table; 3) storing KeyProperties of a HasKey axiom to the class table in which individuals they can uniquely identify.

If a FunctionalProperty P is stored in the table which represents the PropertyDomain C (i.e. in a table $C(\underline{id}, P)$), inserting $\langle x, z\rangle$ and $\langle x, y\rangle$ is not allowed because it violates the PK constraint, so that the UNA is preserved without the need of those triggers defined for the functionality of P. A similar analysis applies for InverseFunctionalProperty and KeyProperties. In general, if we have i functional properties $P_1, \ldots, P_i$, all of which contain a class C as their PropertyDomain, j inverse functional properties $Q_1, \ldots, Q_j$ containing the same C as their PropertyRange, k key properties $R_1, \ldots, R_k$ which together uniquely identify individuals in the same C, then we store the class and properties as a single table:

$$C(\underline{id}, P_1, \ldots, P_i, Q_1, \ldots, Q_j, R_1, \ldots, R_k)$$

Thus facts $P_1(x, a_1), \ldots, P_i(x, a_i)$, $Q_1(b_1, x), \ldots, Q_j(b_j, x)$ and $R_1(x, c_1), \ldots, R_k(x, c_k)$ are stored as one tuple $C(x, a_1, \ldots, a_i, b_1, \ldots, b_j, c_1, \ldots, c_k)$. Moreover, as a result of the schema change, FKs or triggers created on $P_1, \ldots, P_i, Q_1, \ldots, Q_j, R_1, \ldots, R_k$ in the canonical schema are transferred to the table $C(id, P_1, \ldots, P_i, Q_1, \ldots, Q_j, R_1, \ldots, R_k)$.

Take the beer ontology as an example, we may set the property hasColour as a FunctionalProperty

(i.e. one beer can have at most one colour) by either (4.1) or (4.2):

$$\top \sqsubseteq \leqslant 1\, \mathsf{hasColour} \qquad (4.1) \qquad\qquad \mathsf{Fun}(\mathsf{hasColour}) \qquad (4.2)$$

If we additionally recall the PropertyDomain of hasColour, which is defined as Ale or Lager by (2.46), we could map hasColour to an extra column of its PropertyDomain tables (i.e. Ale(id, hasColour) and Lager(id, hasColour)), rather than having three separate tables (i.e. Ale(id), Lager(id) and hasColour(domain, range)) in the canonical schema.

However, since (2.32) makes Beer equivalent to the DisjointUnion of Ale and Lager, a tableaux-based reasoner also infers Beer as the PropertyDomain of hasColour. To build the conventional schema, we will translate hasColour as an extra column of the table Beer rather than Ale or Lager. The reason for this is that the semantics of column hasColour in Beer can be inherited by Ale and Lager because of the FKs $\mathsf{Ale(id)} \overset{\mathsf{fk}}{\Rightarrow} \mathsf{Beer(id)}$ and $\mathsf{Lager(id)} \overset{\mathsf{fk}}{\Rightarrow} \mathsf{Beer(id)}$. In general, if a FunctionalProperty P has two domain classes C and D, and C is more specific than D (i.e. $C \sqsubseteq D$), P is stored as an extra column of the table D representing the more general class D.

Thus if we take the fragment of the beer ontology considered in Section 4.4.1 (i.e. (2.24), (2.25), (2.47) and (2.48)) again, by considering the additional axioms (2.32), (2.46) and (4.1), the canonical schema in Section 4.4.1 can be optimised to the following conventional schema:

$$\mathsf{Ale(id)} \quad \mathsf{Lager(id)} \quad \mathsf{Beer(id, hasColour)}$$

$$\mathsf{hasFlavour(domain, range)} \quad \mathsf{hasDescription(domain, range)}$$

$$\mathsf{Ale(id)} \overset{\mathsf{fk}}{\Rightarrow} \mathsf{Beer(id)} \quad \mathsf{Lager(id)} \overset{\mathsf{fk}}{\Rightarrow} \mathsf{Beer(id)}$$

$$\mathsf{Beer(id, hasColour)} \overset{\mathsf{fk}}{\Rightarrow} \mathsf{hasDescription(domain, range)}$$

$$\mathsf{hasFlavour(domain, range)} \overset{\mathsf{fk}}{\Rightarrow} \mathsf{hasDescription(domain, range)}$$

4.7 Summary

In this chapter we have shown SQOWL2 for providing type inference especially when inserting the A-Box facts of an ontology. SQOWL2 separates T-Box inference and A-Box inference, and we apply a tableaux-based reasoner for the T-Box inference, in order to obtain a complete set of subsumption relations w.r.t. the T-Box. Then, we take the classified T-Box as an input,

and output an ATIDB with tables and triggers. Such an ATIDB is ready to accept any inserts translated from A-Box facts or database users. When data is inserted into a table in ATIDB, triggers associated with this table are automatically invoked to perform type inference. SQOWL2 supports type inference in a transactional and incremental manner, and more specifically, it supports (but is not limited to) the RL profile of the recent OWL 2 release. As a materialised approach, it computes the result of inference before executing queries, and consequently offers fast query processing. However, SQOWL2 is not suitable when the inference closure is extremely large compared with the original A-Box, nor the circumstances in which updates are much more frequent than queries.

Chapter 5

Type Inference from Data Deletes

5.1 Introduction

In Chapter 4, we have detailed how the work SQOWL2 performs type inference when knowledge facts are added as data inserts into the ATIDB. However, to have a comprehensive approach suitable for normal knowledge processing, we must also handle deletes to the knowledge bases. Incrementally updating the inference closure from data deletes is a more difficult problem than handling inserts. First, removing an inferred fact without removing the facts which infer it makes the knowledge base inconsistent. Second, a fact might be inferred in multiple ways, and the fact cannot be removed unless all inference ways are no longer valid. Third, facts might recursively depend on each other, and removing such facts might cause infinite loops.

In this chapter, we provide SQOWL2 which is extended to support type inference from data deletes. The inference materialisation is incrementally updated by implementing a variant of the well-known DRed algorithm in triggers (we call the variant **label & check**). With SQOWL2 to handle inserts and deletes, transactional reasoning is supported. In Section 5.2, we use a motivating example to illustrate what issue of type inference might occur from data deletes. Section 5.3 then provides a review of the DRed algorithm, on which SQOWL2 is based, alongside another deleting algorithm called Counting. Next, in Section 5.4, we illustrate how SQOWL2 deals with the motivating example, and Section 5.5 details triggers generated from OWL 2 RL axioms for handling deletes. Section 5.6 analyses the soundness and completeness

of SQOWL2, and finally Section 5.7 summarises this chapter.

5.2 Motivating Example

To illustrate the problems after facts are deleted from an ontology, we consider a T-Box composed of the axioms (2.24) and (2.28) which have been introduced in Chapter 2, and a new SubClassOf axiom (5.1):

$$\mathsf{Ale} \sqsubseteq \mathsf{Beer} \tag{2.24}$$

$$\mathsf{Beer} \equiv \mathsf{LiquidBread} \tag{2.28}$$

$$\mathsf{IrishBeer} \sqsubseteq \mathsf{LiquidBread} \tag{5.1}$$

in which, (2.24) specifies that every ale is a beer, (2.28) defines that a beer is equivalent to a liquid bread, and (5.1) introduces a new type of liquid bread called IrishBeer. As we have described in Chapter 4, individuals from the four classes in these axioms are stored in tables Ale, Beer, LiquidBread and IrishBeer in the ATIDB. Now, suppose four transactions $T_{5.1}$ – $T_{5.4}$ below are executed in order:

$$T_{5.1} : \mathsf{Ale}(\mathsf{IrishRedAle}) \qquad T_{5.3} : \neg\mathsf{Ale}(\mathsf{IrishRedAle})$$

$$T_{5.2} : \mathsf{IrishBeer}(\mathsf{IrishRedAle}) \qquad T_{5.4} : \neg\mathsf{IrishBeer}(\mathsf{IrishRedAle})$$

where $T_{5.1}$ inserting an A-Box fact Ale(IrishRedAle), $T_{5.2}$ inserting another IrishBeer(IrishRedAle), $T_{5.3}$ deleting the fact Ale(IrishRedAle), and finally $T_{5.4}$ removing IrishBeer(IrishRedAle). The expected changes of the ATIDB is illustrated in Figure 5.1.

The first transaction $T_{5.1}$ should update the ATIDB from the state S_0 to S_1. After inserting the fact Ale(IrishRedAle), because of (2.24) and (2.28), IrishRedAle should be derived not only as an Ale, but also as a Beer and a LiquidBread. Thus, the data of IrishRedAle not only appears in the table Ale, but also in tables Beer and LiquidBread. Here, if we recall the transactional reasoning (i.e. inference results should be available as a part of the atomic action of a database transaction), any other transaction T_c which is concurrent with $T_{5.1}$ should view the ATIDB either as S_0 or S_1, but not any intermediate state. For instance, a sample query Ale(IrishRedAle) $\wedge$ ¬Beer(IrishRedAle) should always evaluate to false in T_c.

The second transaction $T_{5.2}$ should add the fact of IrishBeer(IrishRedAle) into the ATIDB, which

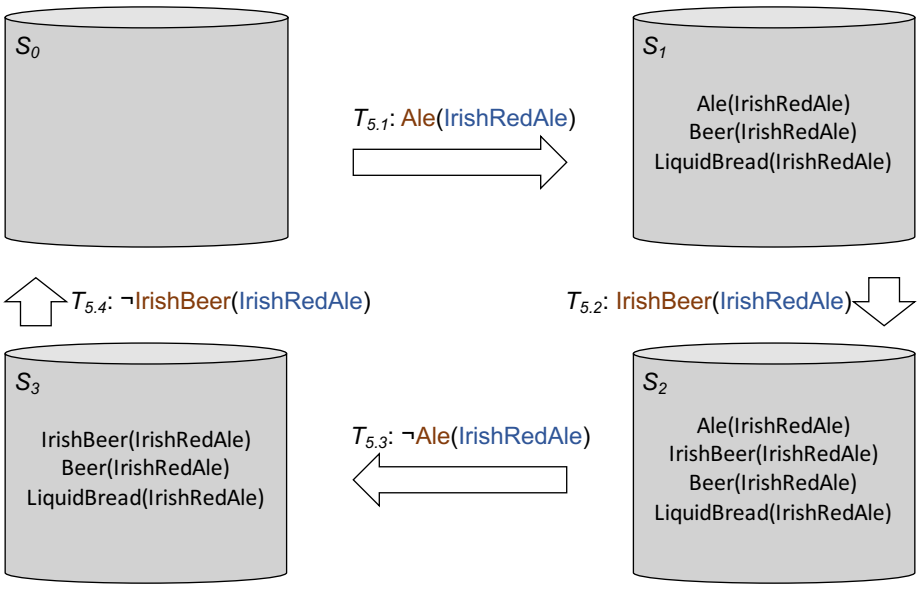

Figure 5.1: Motivating Example of Type Inference from Data Deletes

is then updated to S_2 (i.e. IrishRedAle should be viewed from all the four tables). Note that, from axioms (5.1) and (2.28), IrishRedAle is derived again as a member of LiquidBread and Beer. However, the derivations were already made after executing $T_{5.1}$, and thus should not be added to the ATIDB repeatedly.

The third transaction $T_{5.3}$ removing Ale(IrishRedAle), should only delete IrishRedAle from the table Ale, but not from tables Beer and LiquidBread (i.e. the ATIDB is changed to S_3). Note that although IrishRedAle was inserted into Beer and LiquidBread because of $T_{5.1}$ inserting Ale(IrishRedAle), the two facts can still be derived from IrishBeer(IrishRedAle) and axioms (2.28) and (5.1), even though $T_{5.3}$ removes Ale(IrishRedAle). However, after executing $T_{5.4}$, which deletes IrishBeer(IrishRedAle), the two inferred facts must be removed, and the ATIDB is returned to the empty state.

Moreover, it is important to mention that any attempts at deleting implicit facts, such as deleting Beer(IrishRedAle) or LiquidBread(IrishRedAle) when the ATIDB is in states S_1, S_2 and S_3, should be rejected. Such deletes would make the ATIDB inconsistent, because claiming no such facts exist contradicts what can be inferred from the knowledge base.

5.3 Review of Incremental View Maintenance

Incremental type inference from data inserts and deletes is often known as the problem of incrementally maintaining materialised views in the database field, which has been researched for more than thirty years. As stated in [KM91], the problem of incremental view maintenance is different from **Truth Maintenance** [Doy79] (*a.k.a.* **Belief Revision** [Gär03]) considered in the context of Artificial Intelligence. When new updates alter the existing knowledge, truth maintenance revises some old beliefs to keep the knowledge base consistent. If we consider the motivating example in Section 5.2, when deleting Beer(IrishRedAle) or LiquidBread(IrishRedAle) from S_1, a truth maintenance approach might delete the existing belief Ale(IrishRedAle) to maintain the consistency.

In the context of incremental view maintenance, two well-known algorithms **Counting** and **DRed** described in [GMS93] are commonly adopted by many reasoners [VSM03, UMJ+13, MNPH15]. DRed is known to handle deletes over recursive views, and thus can support OWL 2 RL constructs, such as EquivalentClasses and TransitiveProperty, whilst the Counting algorithm may not correctly deal with recursive views [NY83].

5.3.1 Counting Algorithm

The Counting algorithm introduces for each A-Box fact stored in a database an extra field called **count**, which stores the number of different supports for this fact appearing in the database. In other words, the count represents how many times a fact has been inferred and asserted (here we do not consider repeated explicit facts, i.e. a fact can be explicitly asserted at most once). When loading A-Box facts into a database, the Counting algorithm not only materialises the inference closure, but also keeps accumulating the count of each fact. At the stage of data deleting, some derivations of a fact might be invalid because of the deletions, which causes the algorithm to decrease the count of the fact. When the count of a fact drops to zero, this fact must be removed, since there is no longer any support for the fact to be in the database.

To illustrate the Counting algorithm, take the T-Box axiom (2.24) (i.e. Ale $\sqsubseteq$ Beer) as an

example. Because of this subsumption, for an individual x appears in the class Beer, x needs to be explicitly asserted as a Beer or to be recorded as an Ale (or both of them). This can be expressed as a Datalog rule:

$$\mathsf{Beer}(x) \text{ :- } \mathsf{Ale}(x)$$

Thus if we load two A-Box facts Ale(IrishRedAle) and Beer(IrishRedAle), the views Ale and Beer built by the Counting algorithm should be:

$$\mathsf{Ale} = \{\langle \mathsf{IrishRedAle}, 1\rangle\} \qquad \mathsf{Beer} = \{\langle \mathsf{IrishRedAle}, 2\rangle\}$$

where IrishRedAle in Ale has a count of 1, as only the A-Box fact Ale(IrishRedAle) contributes to the count. The count of IrishRedAle in Beer should be 2, because not only this is explicitly asserted by Beer(IrishRedAle), but also it can be inferred once from IrishRedAle in Ale. Now, if we remove Ale(IrishRedAle), the Counting algorithm decrements the count of IrishRedAle in Ale to 0, which indicates IrishRedAle must be removed from Ale. Due to the absence of IrishRedAle in Ale, one support for holding Beer(IrishRedAle) is lost, which decreases the count of IrishRedAle in Beer by 1. If we continue removing the explicit fact Beer(IrishRedAle), the count of IrishRedAle in Beer should be decrease to 0, which means IrishRedAle must be removed from Beer.

However, the Counting algorithm may not be able to correctly deal with recursive views, as shown in [NY83]. To explain this, consider an additional axiom (2.28) (i.e. Beer $\equiv$ LiquidBread) with the above example. The equivalence relation between Beer and LiquidBread brings another way of inferring facts to Beer (i.e. inferring from LiquidBread besides inferring from Ale). The equivalence also specifies that facts which appear in LiquidBread might be derivations from Beer. We specify this as Datalog rules:

$$\mathsf{Beer}(x) \text{ :- } \mathsf{Ale}(x)$$

$$\mathsf{Beer}(x) \text{ :- } \mathsf{LiquidBread}(x)$$

$$\mathsf{LiquidBread}(x) \text{ :- } \mathsf{Beer}(x)$$

As can be seen, views Beer and LiquidBread recursively depend on each other, which results in infinite loops when building up the count of data items in the two views. Indeed, when the count of x in Beer increments by 1, the count of the same x in LiquidBread should be also increased by 1, which will increase the count of x in Beer by 1 again.

5.3.2 DRed Algorithm

Unlike the Counting algorithm, the DRed algorithm does not store any additional information (e.g. the count for each fact) when loading the explicit facts and materialising the total inference closure. The DRed algorithm contains two phases, which are **over-deletion** and **rederivation**. When an explicit fact is deleted, the over-deletion phase first 'over deletes' all facts which have a derivation from this explicit fact, and then the phase of rederivation 'rederives' some facts which have been over-deleted in the previous phase, but can still be inferred from the remaining facts. One important feature of DRed is this algorithm can handle deletions over recursive views.

To illustrate the DRed algorithm, take the T-Box axioms (2.24), (2.28) and (5.1) in Section 5.2 as an example again. If we further load three A-Box facts Ale(IrishRedAle), IrishBeer(IrishRedAle) and Ale(BritishBrownAle) (asserted by (2.3)), the DRed algorithm will establish views Ale, IrishBeer, Beer and LiquidBread as:

$$\mathsf{Ale} = \{\mathsf{BritishBrownAle}, \mathsf{IrishRedAle}\}$$

$$\mathsf{IrishBeer} = \{\mathsf{IrishRedAle}\}$$

$$\mathsf{Beer} = \{\mathsf{BritishBrownAle}, \mathsf{IrishRedAle}\}$$

$$\mathsf{LiquidBread} = \{\mathsf{BritishBrownAle}, \mathsf{IrishRedAle}\}$$

As can be seen, because of the three explicit A-Box facts, BritishBrownAle and IrishRedAle are present in Ale, and IrishRedAle is recorded in IrishBeer. BritishBrownAle and IrishRedAle appearing in Beer and Liquid are because of inference. Since DRed does not need to store the count of each fact, the infinite loop experienced by the Counting algorithm is avoided here, though Beer and LiquidBread recursively infer facts to each other.

Now suppose we remove the A-Box fact Ale(IrishRedAle), DRed not only deletes IrishRedAle from Ale, but also 'over deletes' IrishRedAle from Beer and LiquidBread, because the deleted fact can infer the over-deleted facts. Then, the rederivation phase computes as to whether the deleted IrishRedAle in Ale, Beer and LiquidBread can still be inferred from the remaining facts. Therefore, IrishRedAle is rederived into Beer and LiquidBread, as IrishRedAle in IrishBeer can still infer them. If we further remove the fact IrishBeer(IrishRedAle), IrishRedAle in IrishBeer, Beer

and LiquidBread is 'over deleted' first; however, none of them can be rederived.

The two algorithms have their advantages and disadvantages. From the viewpoint of inserting facts, the Counting algorithm needs to not only infer and store implicit facts, but also to increment the count of each fact. However, the DRed only materialises the inference closure without storing such additional information. At the stage of deleting facts, the Counting algorithm simply decrements the count of affected facts, and removes them if their counts drop to zero; by contrast, the DRed algorithm requires to perform two phases to handle the deletes, which is less efficient. Another key difference between the two algorithms is that DRed is known to support deletes over recursive views, while the Counting algorithm does not. Thus DRed can be used to handle OWL 2 RL axioms, such as EquivalentClasses and TransitiveProperty.

5.4 SQOWL2 for Data Deletes

In this section, we show how SQOWL2 in Chapter 4 can be extended to additionally support deletes. Following the same approach architecture shown in Figure 4.4, for a classified T-Box obtained by a tableaux-based reasoner, SQOWL2 establishes an ATIDB with tables and triggers, which is able to perform type inference in a transactional and incremental manner. Tables in the ATIDB are either unary relations mapped from OWL classes or binary relations translated from OWL properties. Triggers in the ATIDB are created from OWL axioms and demonstrated as ECA rules of the form **when** ⟨event⟩ **if** ⟨condition⟩ **then** ⟨action⟩. The ATIDB is able to load the A-Box data as inserts, and moreover, accepts additions and deletions to the A-Box launched by database users expressed as updates on the tables which denote the classes and properties.

In particular for handling data deletes, which is the new feature compared to Chapter 4, the triggers automatically launch a **label & check** process, which first recursively labels all data items that have a derivation from the data in the deletes. Then the second phase checks as to whether the labelled data can be inferred or not from non-labelled data; if so we keep this data; otherwise, we remove it. In order to perform the process of label & check, we introduce

a state for each data item stored in the ATIDB (described in Section 5.4.1), and identify two types of inserts/deletes, namely ontology and reasoner inserts/deletes (in Section 5.4.2). Then, in Section 5.4.3, we use the motivating example to show how the new triggers materialise the inference closure with the consideration of the data state, when $T_{5.1}$ and $T_{5.2}$ are executed over the ATIDB. Afterwards, Section 5.4.4 demonstrates how the label & check process is performed for handling $T_{5.3}$ and $T_{5.4}$ in the motivating example.

5.4.1 The State of Each Data Item

In the ATIDB, SQOWL2 views each individual x in a class C as an item of data denoted as $C(x)$, and each data item is specified as passing through four states (i.e. ø, e, i and d) by certain database operations, as shown in Figure 5.2. Similarly, a property fact $P(x, y)$ is viewed as an item of data $P(x, y)$, which passes through the same four states in the ATIDB.

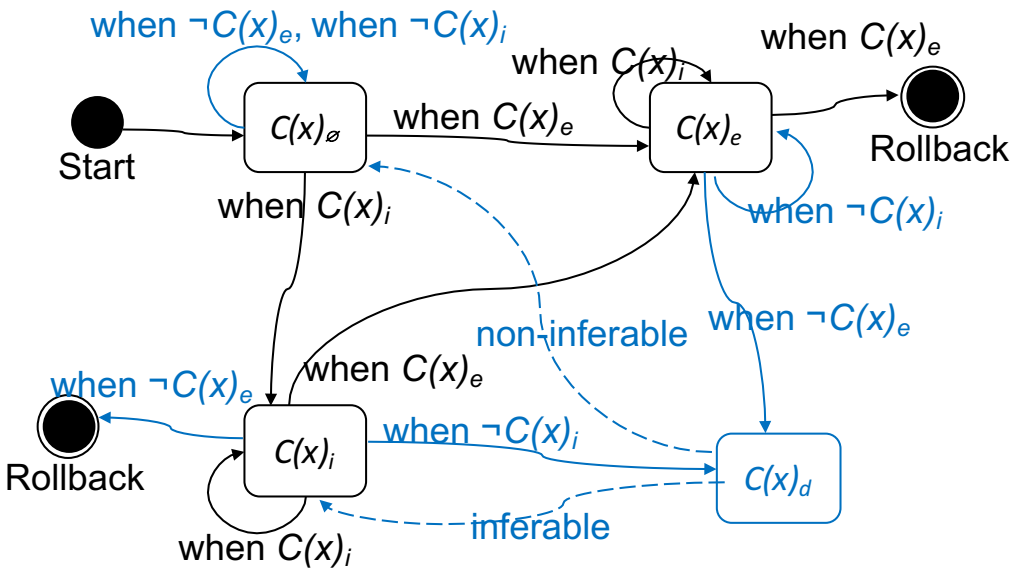

Figure 5.2: Data state transitions

In particular, the data item $C(x)$ with each state of ø, e, i and d has the following semantics:

- $C(x)_{ø}$: a class fact which does not hold, and therefore is not yet materialised in the table C, which represents the class C, in the ATIDB

- $C(x)_e$: a class fact which is materialised in the ATIDB, because an explicit A-Box fact defines it
- $C(x)_i$: a class fact which is materialised in the ATIDB, since it can be implicitly inferred from other facts
- $C(x)_d$: one of the supporting arguments for storing a class fact in the ATIDB has been lost, and the check phase which determines whether the fact is still inferable from other facts should be launched

5.4.2 Ontology Inserts/Deletes & Reasoner Inserts/Deletes

Each of the four states can be transitioned to another by insert or delete operations. We classify these operations into two categories, which are **ontology inserts/deletes** and **reasoner inserts/deletes**:

- An **ontology insert** describes an event that some user or application is inserting into the ATIDB a new explicit fact, and such an insert is captured by the trigger **when** $C(x)_e$
- A **reasoner insert** identifies an event that some implicit facts have been inferred by a reasoner from the existing facts, and the reasoner has attempted to insert these implicit facts into the ATIDB. A reasoner insert is detected by the trigger **when** $C(x)_i$
- An **ontology delete** is an event in which some user or application is deleting an explicit fact from the ATIDB, and such a delete is captured by **when** $\neg C(x)_e$
- A **reasoner delete** describes an event that when some evidences for persisting a fact in the ATIDB have been removed, detected by the trigger **when** $\neg C(x)_i$

The two categories of operations change the state of each data item, and possible state transitions are also outlined in Figure 5.2. Firstly, for ontology or reasoner inserts:

- For $C(x)_\emptyset$, which denotes a fact not stored in the ATIDB, an ontology insert of $C(x)_e$ transitions $C(x)_\emptyset$ to $C(x)_e$, and a reasoner insert of $C(x)_i$ updates $C(x)_\emptyset$ to $C(x)_i$. The

state ø is essentially a logical denotation which we mainly use for ease of expressing a fact absent from the ATIDB, and it is unnecessary to explicitly store this state from the viewpoint of implementation (shown in Chapter 7).

- For $C(x)_i$, which identifies a fact materialised with the implicit state, repeated reasoner inserts of $C(x)_i$ do not modify the state, in order to avoid duplicated derivations which are based on $C(x)_i$. However, an ontology insert of $C(x)_e$ gives the fact explicit semantics, which updates $C(x)_i$ to $C(x)_e$.

- For $C(x)_e$, which means a fact stored with the explicit state, a reasoner insert of $C(x)_i$ does not modify the state, so that duplicated inference is prevented. Here, in order to implement normal database semantics, attempting further ontology inserts of $C(x)_e$ results in a rollback.

Secondly, for ontology or reasoner deletes:

- For $C(x)_\emptyset$, which logically denotes an absent data item, ontology or reasoner deletes are both ignored, in order to match the normal database semantics that deleting data which is not present causes no errors.

- For $C(x)_i$, executing an ontology delete $\neg C(x)_e$ results in the ATIDB entering into an inconsistent state, because being no $C(x)$ by the ontology delete conflicts with what can be derived from other data stored in the ATIDB, and such an ontology delete should be rolled back. However, a reasoner delete $\neg C(x)_i$ updates $C(x)_i$ to $C(x)_d$, in order to label the data for checking, as it might still be inferable from other items of data.

- For $C(x)_e$, an ontology delete $\neg C(x)_e$ modifies it to $C(x)_d$, since the data might still be derivable even after deleting its explicit supporting argument. However, a reasoner delete $\neg C(x)_i$ does not lead to any change, because only ontology deletes are able to remove the explicit semantics.

- For the state of $C(x)$ is changed to d (i.e. the fact is labelled), a further recursive labelling process is launched to attempt reasoner deletes over other data items which depend on

$C(x)_d$. After the labelling process is finished, all data items with d are subject to a check process, which determines whether they can still be inferred from non-labelled data (i.e. data items in state e or i). If so, we update $C(x)_d$ to $C(x)_i$; otherwise, $C(x)_d$ is removed from the ATIDB (i.e. $C(x)_d$ is logically changed to $C(x)_\emptyset$).

5.4.3 Materialising Type Inference with Data State

Considering the data state introduced for enabling SQOWL2 to handle deletes, when A-Box facts are loaded into the ATIDB, not only type inference should be computed and materialised, but also the state of each data should be correctly identified. Thus, the triggers which we have defined in Chapter 4 should be improved, and here we show the improvements by using the motivating example in Section 5.2.

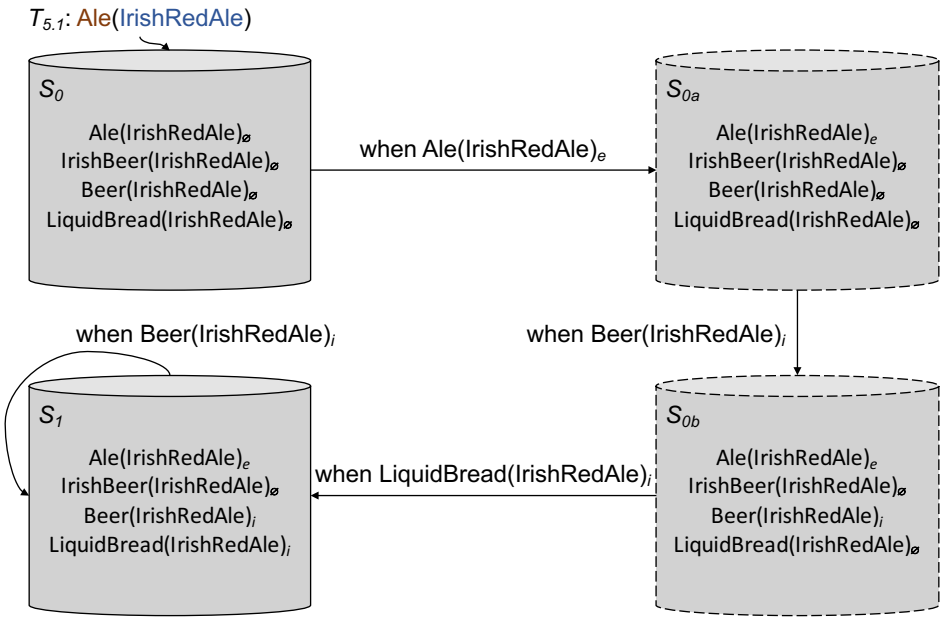

Figure 5.3: $T_{5.1}$: insert Ale(IrishRedAle)

Figure 5.3 gives the detailed execution process of $T_{5.1}$ shown as an atomic transition in Figure 5.1. It shows that $T_{5.1}$ changes the ATIDB from S_0 to S_1 via two intermediate states S_{0_a} and S_{0_b}. Firstly, inserting Ale(IrishRedAle) (i.e. an ontology insert of $\mathsf{Ale}(\mathsf{IrishRedAle})_e$) into the ATIDB S_0 is checked by a before trigger:

$$\textbf{when } {}^{-}\mathsf{Ale}(x)_e \textbf{ if } \neg\mathsf{Ale}(x)_e \textbf{ then } \mathsf{Ale}(x)_e$$

Because $\mathsf{Ale}(\mathsf{IrishRedAle})_{\emptyset}$ is true (the data is not yet stored in the ATIDB S_0), the insert is therefore permitted, and updates the ATIDB to S_{0_a}. The SubClassOf axiom (2.24) is translated into an after trigger:

$$\textbf{when } {}^{+}\mathsf{Ale}(x)_{e \vee i} \textbf{ then } \mathsf{Beer}(x)_i$$

Thus after $\mathsf{Ale}(\mathsf{IrishRedAle})_e$ is materialised, this after trigger is fired to attempt a reasoner insert of $\mathsf{Beer}(\mathsf{IrishRedAle})_i$, which is detected by a before trigger created on Beer:

$$\textbf{when } {}^{-}\mathsf{Beer}(x)_i \textbf{ if } \neg\mathsf{Beer}(x)_i \textbf{ then } \mathsf{Beer}(x)_i$$

Because $\mathsf{Beer}(\mathsf{IrishRedAle})_{\emptyset}$ is true in S_{0_a}, the reasoner insert of $\mathsf{Beer}(\mathsf{IrishRedAle})_i$ is allowed by the above before trigger, and the ATIDB is updated from S_{0_a} to S_{0_b}. An EquivalentClasses axiom can be logically treated as two SubClassOf axioms; therefore, the axiom (2.28) is mapped into two after triggers:

$$\textbf{when } {}^{+}\mathsf{Beer}(x)_{e \vee i} \textbf{ then } \mathsf{LiquidBread}(x)_i$$

$$\textbf{when } {}^{+}\mathsf{LiquidBread}(x)_{e \vee i} \textbf{ then } \mathsf{Beer}(x)_i$$

After the ATIDB is changed to S_{0_b}, materialising $\mathsf{Beer}(\mathsf{IrishRedAle})_i$ causes another reasoner insert of $\mathsf{LiquidBread}(\mathsf{IrishRedAle})_i$, which updates the state of $\mathsf{LiquidBread}(\mathsf{IrishRedAle})$ from $\emptyset$ to i (i.e. S_{0_b} is updated to S_1), because of the following before trigger on LiquidBread:

$$\textbf{when } {}^{-}\mathsf{LiquidBread}(x)_i \textbf{ if } \neg\mathsf{LiquidBread}(x)_i \textbf{ then } \mathsf{LiquidBread}(x)_i$$

Note that after storing $\mathsf{LiquidBread}(\mathsf{IrishRedAle})_i$, the after insert trigger on LiquidBread generates the reasoner insert of $\mathsf{Beer}(\mathsf{IrishRedAle})_i$ again. However, this repeated reasoner insert is ignored, because $\mathsf{Beer}(\mathsf{IrishRedAle})_i$ is already materialised in S_1 (i.e. the ATIDB stays at S_1), so that duplicated inference is avoided.

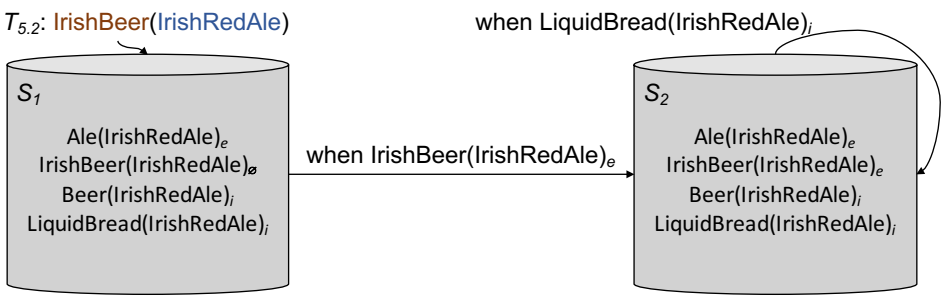

Figure 5.4: $T_{5.2}$: insert IrishBeer(IrishRedAle)

Figure 5.4 illustrates the execution process of $T_{5.2}$, inserting IrishBeer(IrishRedAle). $T_{5.2}$ is treated as an ontology insert of IrishBeer(IrishRedAle)$_e$, which changes the state of IrishBeer(IrishRedAle) from ø to e (i.e. S_1 is updated to S_2), as IrishBeer(IrishRedAle)$_\emptyset$ is true in S_1. After this, the after insert trigger on IrishBeer from the axiom (5.1):

$$\textbf{when } {}^{+}\mathsf{IrishBeer}(x)_{e \vee i} \textbf{ then } \mathsf{LiquidBread}(x)_i$$

attempts a new reasoner insert of LiquidBread(IrishRedAle)$_i$. As LiquidBread(IrishRedAle)$_i$ is already true in S_2, such a reasoner insert is ignored by the before insert trigger on LiquidBread (i.e. the ATIDB remains at S_2).

5.4.4 Label & Check for Data Deletes

Executing an ontology delete over an item of data in state i without deleting the explicit facts which infer the data causes inconsistencies in the ATIDB. Therefore, such ontology deletes should be rolled back, and we specify for Beer and LiquidBread the following two before triggers:

$$\textbf{when } {}^{-}\neg\mathsf{Beer}(x)_e \textbf{ if } \mathsf{Beer}(x)_i \textbf{ then } \mathsf{rollback}$$

$$\textbf{when } {}^{-}\neg\mathsf{LiquidBread}(x)_e \textbf{ if } \mathsf{LiquidBread}(x)_i \textbf{ then } \mathsf{rollback}$$

When the ATIDB is in S_0, S_1 or S_2, the above triggers result in executing an ontology delete to Beer(IrishRedAle)$_i$ or to LiquidBread(IrishRedAle)$_i$ to rolling back the deletion. In essence, we only permit attempting an ontology delete to data in state e, i.e. $C(x)_e$ or $P(x,y)_e$. However, executing ontology deletes over $C(x)_e$ or $P(x,y)_e$ only removes the explicit semantics, and because they might also be implicitly derived, a label & check process detailed below should be launched to verify whether they can still be derived.

Label: when a user or application attempts to delete $C(x)_e$, we create a before delete trigger on C to convert this ontology delete to an update of $C(x)_e$ to $C(x)_d$. This update **labels** $C(x)$ in state d, and the label process might cascade to other data which depends on $C(x)$ by attempting reasoner deletes to data inferred from $C(x)_e$. For example, as (2.24) means that individuals in Ale infer the same in Beer, when a data item x in Ale is labelled, a reasoner delete to the same x in Beer should be conducted by the following after trigger created on Ale:

$$\textbf{when } {}^{+}\mathsf{Ale}(x)_d \textbf{ then } \neg\mathsf{Beer}(x)_i$$

This reasoner delete is then handled by a before delete trigger created on Beer shown as follows:

$$\textbf{when } {}^{-}\neg\mathsf{Beer}(x)_i \textbf{ if } \mathsf{Beer}(x)_i \textbf{ then } \mathsf{Beer}(x)_d$$

As can be seen, if $\mathsf{Beer}(x)_i$ is true, attempting a reasoner delete to x in Beer will be converted as an update of $\mathsf{Beer}(x)_i$ to $\mathsf{Beer}(x)_d$, i.e. the labelling process cascades. Note that, the after delete trigger on Ale and the before delete trigger on Beer can be merged as a single after delete trigger on Ale shown as:

$$\textbf{when } {}^{+}\mathsf{Ale}(x)_d \textbf{ if } \mathsf{Beer}(x)_i \textbf{ then } \mathsf{Beer}(x)_d$$

Analogous to this trigger, the axiom (5.1), which expresses a subsumption relation from IrishBeer to LiquidBread, generates another after delete trigger on IrishBeer as:

$$\textbf{when } {}^{+}\mathsf{IrishBeer}(x)_d \textbf{ if } \mathsf{LiquidBread}(x)_i \textbf{ then } \mathsf{LiquidBread}(x)_d$$

Moreover, the EquivalentClasses axiom (2.28)) is translated into two after delete triggers on tables Beer and LiquidBread:

$$\textbf{when } {}^{+}\mathsf{Beer}(x)_d \textbf{ if } \mathsf{LiquidBread}(x)_i \textbf{ then } \mathsf{LiquidBread}(x)_d$$

$$\textbf{when } {}^{+}\mathsf{LiquidBread}(x)_d \textbf{ if } \mathsf{Beer}(x)_i \textbf{ then } \mathsf{Beer}(x)_d$$

We now take $T_{5.3}$, which removes Ale(IrishRedAle), as an example to show how the label process cascades to all inferred facts from Ale(IrishRedAle); a detailed execution is given in Figure 5.5.

$T_{5.3}$, which conducts an ontology delete to Ale(IrishRedAle), first changes $\mathsf{Ale}(\mathsf{IrishRedAle})_e$ to $\mathsf{Ale}(\mathsf{IrishRedAle})_d$ moving the ATIDB from S_2 to S_{2_a}. This labelling generates a reasoner delete of $\mathsf{Beer}(\mathsf{IrishRedAle})_i$, which updates $\mathsf{Beer}(\mathsf{IrishRedAle})_i$ to $\mathsf{Beer}(\mathsf{IrishRedAle})_d$, and the ATIDB is consequently changed to S_{2_b}. Afterwards, the after delete trigger created on Beer attempts a reasoner delete of $\mathsf{LiquidBread}(\mathsf{IrishRedAle})_i$, which changes the data state to d, and the ATIDB enters S_{2_c} (i.e. the labelling process because of $T_{5.3}$ removing Ale(IrishRedAle) is finished).

Check: All labelled data items (i.e. in state d) are checked as to whether they are still implicitly inferable or not from non-labelled data items (i.e. stated in e or i). For labelled data which is still inferable, its state d is changed to i; otherwise, the data is removed (which is logically

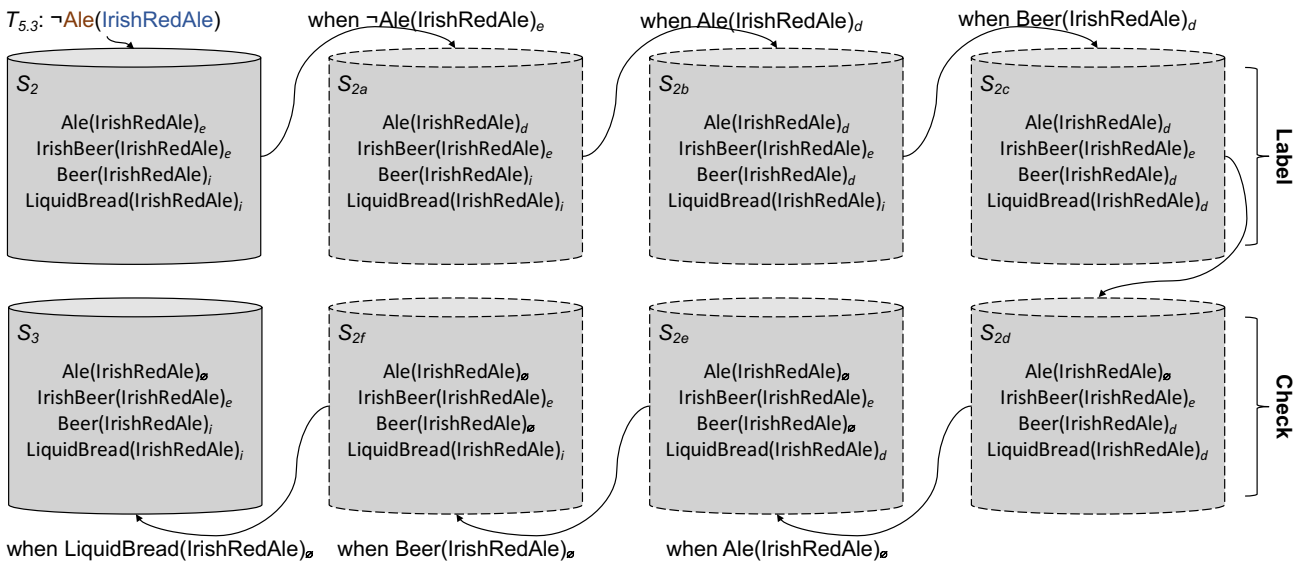

Figure 5.5: $T_{5.3}$: delete Ale(IrishRedAle)

interpreted as changing d to $\emptyset$). To realise the check process, we create for each table a Datalog-style **inference rule** containing all inference logic to the table (denoted as $C_{Int}(x)$ for a class table or $P_{Int}(x, y)$ for a property table). Next, the check is processed by invoking the following trigger created on each table (the "$\oplus$" symbol specifies the check phase should start after all the labelling has finished):

$$\textbf{when } {}^{\oplus}C(x)_d \textbf{ if } C_{Int}(x) \textbf{ then } C(x)_i \textbf{ else } C(x)_\emptyset$$

$$\textbf{when } {}^{\oplus}P(x,y)_d \textbf{ if } P_{Int}(x,y) \textbf{ then } P(x,y)_i \textbf{ else } P(x,y)_\emptyset$$

Note that, the inference rule for a table is omitted, if the table contains no inference logic, such as the case for tables Ale and IrishBeer when only taking axioms (2.24), (2.28) and (5.1) into account. However, for tables Beer and LiquidBread, their inference rules includes their subclasses and equivalent classes:

$$\mathsf{Beer}_{Int}(x) \text{ :- } \mathsf{Ale}(x)_{e \vee i} \qquad \mathsf{LiquidBread}_{Int}(x) \text{ :- } \mathsf{IrishBeer}(x)_{e \vee i}$$

$$\mathsf{Beer}_{Int}(x) \text{ :- } \mathsf{LiquidBread}(x)_{e \vee i} \qquad \mathsf{LiquidBread}_{Int}(x) \text{ :- } \mathsf{Beer}(x)_{e \vee i}$$

Continuing the execution of $T_{5.3}$ in Figure 5.5, the check phase starts from $\mathsf{Ale(IrishRedAle)}_d$ in S_{2_c}. As $\mathsf{Ale}_{Int}(x)$ is empty, $\mathsf{Ale(IrishRedAle)}_d$ is updated to $\mathsf{Ale(IrishRedAle)}_\emptyset$ (i.e. Ale(IrishRedAle) is removed), and the ATIDB enters S_{2_d}. $\mathsf{Beer(IrishRedAle)}_d$ is checked next, and because the data IrishRedAle in Ale is in state $\emptyset$ and in LiquidBread is in state d, it is updated to $\mathsf{Beer(IrishRedAle)}_\emptyset$, which causes the ATIDB to change from S_{2_d} to S_{2_e}. Then, $\mathsf{LiquidBread(IrishRedAle)}_d$ is checked

and updated with the implicit state i because of $\mathsf{IrishBeer}(\mathsf{IrishRedAle})_e$ (i.e. the ATIDB moves from S_{2_e} to S_{2_f}). Due to the occurrence of $\mathsf{LiquidBread}(\mathsf{IrishRedAle})_i$, the after insert trigger on LiquidBread for conducting a reasoner insert of $\mathsf{Beer}(\mathsf{IrishRedAle})_i$ is invoked, and changes the ATIDB from S_{2_f} to S_3, ending the whole label & check process for $T_{5.3}$. Analogous to processing $T_{5.3}$, applying the label & check process to $T_{5.4}$ changes the ATIDB back to the empty state S_0.

5.5 Triggers and Inference Rules for OWL 2 RL axioms

In the previous section, we have demonstrated how SQOWL2 performs type inference from data updates, especially from data deletes, by using the motivating example. In fact, not only SubClassOf and EquivalentClasses axioms in the motivating example, but also all the other OWL 2 RL axioms are covered by SQOWL2. Note that, because the UNA is adopted by our approach, we do not allow the use of the constructor SameIndividual, which implies that one knowledge object shares two individual names. To avoid verbose description, we choose some typical constructors in OWL 2 RL to illustrate how triggers and inference rules are generated, and summarise how other RL constructors are implemented in Table 5.1 and 5.2.

5.5.1 OWL Classes and Properties

As demonstrated in Figure 5.2, ontology and reasoner inserts/deletes change a data item in the ATIDB passing though four data states. The data transitions are achieved by invoking before triggers. Firstly, for ontology and reasoner inserts, before insert triggers for every class table are specified as follows:

$$\begin{aligned}
\text{Class: } C \rightsquigarrow \ & \textbf{when } {}^{-}C(x)_e \ \ \textbf{if } C(x)_{\emptyset \vee i} \textbf{ then } C(x)_e \\
& \qquad\qquad\qquad\quad \textbf{if } C(x)_e \textbf{ then } \mathsf{rollback} \\
& \textbf{when } {}^{-}C(x)_i \ \ \textbf{if } C(x)_{\emptyset} \textbf{ then } C(x)_i \\
& \qquad\qquad\qquad\quad \textbf{if } C(x)_{e \vee i} \textbf{ then } \mathsf{ignore}
\end{aligned}$$

and very similarly, before insert triggers on a property table are specified as follows:

$$\begin{aligned}
\text{Property: } P \rightsquigarrow \quad & \textbf{when } {}^{-}P(x,y)_e && \textbf{if } P(x,y)_{\emptyset \vee i} \textbf{ then } P(x,y)_e \\
& && \textbf{if } P(x,y)_e \textbf{ then } \mathsf{rollback} \\
& \textbf{when } {}^{-}P(x,y)_i && \textbf{if } P(x,y)_{\emptyset} \textbf{ then } P(x,y)_i \\
& && \textbf{if } P(x,y)_{e \vee i} \textbf{ then } \mathsf{ignore}
\end{aligned}$$

Ontology inserts are detected by **when** ${}^{-}C(x)_e$ or **when** ${}^{-}P(x,y)_e$, and will be able to change the state of an item of data from state $\emptyset$ or i to state e, in order to give the data explicit semantics. An ontology insert should be rolled back if a data item is already in state e, to match the normal database semantics which rejects a repeated insert of the same data. However, for reasoner inserts, which are detected by **when** ${}^{-}C(x)_i$ or **when** ${}^{-}P(x,y)_i$, the triggers will be able to change the state of data to i only when the data is in state $\emptyset$; when the data is in state e or i the before triggers will ignore the reasoner inserts, since the inserted data is already known to the ATIDB.

Secondly, for ontology and reasoner deletes, we create before delete triggers to ignore, rollback or label the data which is intended for deletion (i.e. update the data to be in state d). In particular, the before triggers for handling deletes over a class table are:

$$\begin{aligned}
\text{Class: } C \rightsquigarrow \quad & \textbf{when } {}^{-}\neg C(x)_e && \textbf{if } C(x)_{\emptyset} \textbf{ then } \mathsf{ignore} \\
& && \textbf{if } C(x)_i \textbf{ then } \mathsf{rollback} \\
& && \textbf{if } C(x)_e \textbf{ then } C(x)_d \\
& \textbf{when } {}^{-}\neg C(x)_i && \textbf{if } C(x)_{\emptyset \vee e} \textbf{ then } \mathsf{ignore} \\
& && \textbf{if } C(x)_i \textbf{ then } C(x)_d
\end{aligned}$$

The triggers detect ontology deletes over a class table by **when** ${}^{-}\neg C(x)_e$, which can then update a data item in state e to d, and capture reasoner deletes by **when** ${}^{-}\neg C(x)_i$, which can then change an item of data in state i to d. Ontology deletes will be ignored, if the data being deleted is not present (i.e. in state $\emptyset$), or will be rolled back if they attempt to delete data in state i (preventing the ATIDB entering inconsistent state). Moreover, reasoner deletes will be ignored if the data is in state $\emptyset$ or e. Before triggers for ontology or reasoner deletes over property tables are similar:

$$\begin{aligned}\text{Property: } P \rightsquigarrow \ & \textbf{when } {}^{-}\neg P(x,y)_e \ \ \textbf{if } P(x,y)_\emptyset \textbf{ then ignore} \\ & \qquad\qquad\qquad\quad \textbf{if } P(x,y)_i \textbf{ then rollback} \\ & \qquad\qquad\qquad\quad \textbf{if } P(x,y)_e \textbf{ then } P(x,y)_d \\ & \textbf{when } {}^{-}\neg P(x,y)_i \ \ \textbf{if } P(x,y)_{\emptyset\vee e} \textbf{ then ignore} \\ & \qquad\qquad\qquad\quad \textbf{if } P(x,y)_i \textbf{ then } P(x,y)_d\end{aligned}$$

Note that all the triggers above have the purpose of labelling data, possibly cause cascade of labelling some other data items. When the label process is finished over all affected data items, the next step is to check as to whether the labelled data is still inferable. This is conducted by an after trigger created on each class and property table:

$$\text{Class: } C \rightsquigarrow \textbf{when } {}^{\oplus}C(x)_d \textbf{ if } C_{Int}(x) \textbf{ then } C(x)_i \textbf{ else } C(x)_\emptyset$$

$$\text{Property: } P \rightsquigarrow \textbf{when } {}^{\oplus}P(x,y)_d \textbf{ if } P_{Int}(x,y) \textbf{ then } P(x,y)_i \textbf{ else } P(x,y)_\emptyset$$

Recall that these triggers perform checking after all the labelling has been finished as specified by the "$^{\oplus}$" symbol. If an item of data is still inferable from the inference rule for this table, it will be updated to be in state i; otherwise, it will be removed from the ATIDB (i.e. the state is updated to $\emptyset$).

5.5.2 The Semantics of Class Axioms

Apart from constructors SubClassOf and EquivalentClass, the handling of which has already been illustrated in Section 5.4, we choose another four typical constructors DisjointClasses, AllValuesFrom, IntersectionOf, and OneOf to show how other class-related constructors in OWL 2 RL can be mapped to triggers and inference rules. The handling of the remaining class-related constructors are listed in Table 5.1.

DisjointClasses: OWL 2 RL uses DisjointClasses to state that several classes do not share the same individual as exemplified by the axiom (2.29), which specifies that Ale and Lager are disjoint with each other. As we have explained, a before trigger should be created on each of the DisjointClasses tables, to prevent inconsistencies in the ATIDB. Here, since we introduce the state of the data, the transformation rule for mapping DisCla($C_1, \ldots, C_n$) into triggers should

Table 5.1: Triggers and inference rules generated from class axioms

DL Syntax	Triggers	Inference Rules
$\exists P.C_E \sqsubseteq C$	**when** ${}^{+}P(x,y)_{e\vee i}$ **if** $C_E(y)_{e\vee i}$ **then** $C(x)_i$ **when** ${}^{+}C_E(y)_{e\vee i}$ **if** $P(x,y)_{e\vee i}$ **then** $C(x)_i$ **when** ${}^{+}P(x,y)_d$ **if** $C_E(y)_{e\vee i\vee d}$ **then** $\neg C(x)_i$ **when** ${}^{+}C_E(y)_d$ **if** $P(x,y)_{e\vee i\vee d}$ **then** $\neg C(x)_i$	$C_{Int}(x)$:- $P(x,y)_{e\vee i}, C_E(y)_{e\vee i}$
$C \sqsubseteq \exists P.\{a\}$	**when** ${}^{+}C(x)_{e\vee i}$ **then** $P(x,a)_i$ **when** ${}^{+}C(x)_d$ **then** $\neg P(x,a)_i$	$P_{Int}(x,a)$:- $C(x)_{e\vee i}$
$\exists P.\{a\} \sqsubseteq C$	**when** ${}^{+}P(x,a)_{e\vee i}$ **then** $C(x)_i$ **when** ${}^{+}P(x,a)_d$ **then** $\neg C(x)_i$	$C_{Int}(x)$:- $P(x,a)_{e\vee i}$
$C \sqsubseteq \exists P.\mathsf{Self}$	**when** ${}^{+}C(x)_{e\vee i}$ **then** $P(x,x)_i$ **when** ${}^{+}C(x)_d$ **then** $\neg P(x,x)_i$	$P_{Int}(x,x)$:- $C(x)_{e\vee i}$
$\exists P.\mathsf{Self} \sqsubseteq C$	**when** ${}^{+}P(x,x)_{e\vee i}$ **then** $C(x)_i$ **when** ${}^{+}P(x,x)_d$ **then** $\neg C(x)_i$	$C_{Int}(x)$:- $P(x,x)_{e\vee i}$
$\geqslant n\, P \sqsubseteq C$	**when** ${}^{+}P(x,y)_{e\vee i}$ **if** $count[P(x,\_)_{e\vee i}]\geq n$ **then** $C(x)_i$ **when** ${}^{+}P(x,y)_d$ **if** $count[P(x,\_)_{e\vee i\vee d}]\geq n$ **then** $\neg C(x)_i$	$C_{Int}(x)$:- $count[P(x,\_)_{e\vee i}]\geq n$
$\geqslant n\, P.C_E \sqsubseteq C$	**when** ${}^{+}P(x,y)_{e\vee i}$ **if** $count[P(x,y)_{e\vee i}, C_E(y)_{e\vee i}]\geq n$ **then** $C(x)_i$ **when** ${}^{+}C_E(y)_{e\vee i}$ **if** $count[P(x,y)_{e\vee i}, C_E(y)_{e\vee i}]\geq n$ **then** $C(x)_i$ **when** ${}^{+}P(x,y)_d$ **if** $count[P(x,y)_{e\vee i\vee d}, C_E(y)_{e\vee i\vee d}]\geq n$ **then** $\neg C(x)_i$ **when** ${}^{+}C_E(y)_d$ **if** $count[P(x,y)_{e\vee i\vee d}, C_E(y)_{e\vee i\vee d}]\geq n$ **then** $\neg C(x)_i$	$C_{Int}(x)$:- $count[P(x,y)_{e\vee i}, C_E(y)_{e\vee i}]\geq n$

be refined to deal with the state of data as:

$$\mathsf{DisCla}(C_1,\ldots,C_n) \rightsquigarrow \textbf{when } {}^{-}C_i(x)_{e\vee i}\ 1\leq i\leq n$$
$$\textbf{if } C_1(x)_{e\vee i} \text{ or } \ldots \text{ or } C_{i-1}(x)_{e\vee i} \text{ or } C_{i+1}(x)_{e\vee i} \text{ or } \ldots \text{ or } C_n(x)_{e\vee i}$$
$$\textbf{then } \mathsf{rollback}$$

Thus, ontology or reasoner inserts to a class table are rolled back if the data of the inserts already exists in state e or i in one of the other disjoint-class tables. Also as can be seen from the above triggers, no reasoner insert is generated, which means no implicit fact can be derived from DisjointClasses axioms. Therefore, the inference rule for the table which represents each of the disjoint classes should be empty, and when a data item in one of the tables is labelled, the label process will not cascade because of the semantics of DisjointClasses.

Note that compared to the triggers for handling inserts illustrated in Chapter 4, the refined triggers need to detect ontology or reasoner inserts (i.e. **when** $C(x)_{e\vee i}$). Another difference is

in the ⟨condition⟩ field, the refined trigger needs to verify the condition over data in state e or i, which is unnecessary for triggers in Chapter 4.

AllValuesFrom: as we have reviewed in Chapter 2, denotes itself in DL as $\forall P.C_E$, which specifies a set of individuals x, such that if x is related to some y by a property P, then the individuals y always belong to C_E. The RL profile only allows using $\forall P.C_E$ as a super-class expression (i.e. $C \sqsubseteq \forall P.C_E$), because using it as a subclass expression does not infer anything, as we analysed in Section 4.5.2. For $C \sqsubseteq \forall P.C_E$, triggers for inserts are refined as:

$$\text{AllValuesFrom: } C \sqsubseteq \forall P.C_E \rightsquigarrow \textbf{when } {}^{+}C(x)_{e\vee i} \textbf{ if } P(x,y)_{e\vee i} \textbf{ then } C_E(y)_i$$
$$\textbf{when } {}^{+}P(x,y)_{e\vee i} \textbf{ if } C(x)_{e\vee i} \textbf{ then } C_E(y)_i$$

Recall that $C \sqsubseteq \forall P.C_E$ implies that for any individual x of C, if x appears in $\langle x, y\rangle$ of P, then y should be derived as an instance of C_E. Therefore, the trigger on the table C will detect the ontology or reasoner inserts of x (i.e. **when** ${}^{+}C(x)_{e\vee i}$), and after these inserts have been executed, if $\langle x, y\rangle$ is in state e or i and P (i.e. **if** $P(x,y)_{e\vee i}$), then a reasoner insert of $C_E(y)_i$ should be attempted (i.e. **then** $C_E(y)_i$). Similarly, the reasoner insert of $C_E(y)_i$ should also be generated by another trigger on the table P, after ontology or reasoner inserts of $\langle x, y\rangle$ to P (i.e. **when** $P(x,y)_{e\vee i}$) have been executed, and $C(x)$ is in state e or i. Note that, as we have clarified in Section 4.5.1, free variables in the field of ⟨condition⟩, such as the y which appears in **if** $P(x,y)_{e\vee i}$, are universally quantified.

When considering the label phase of handling deletes, two triggers C and P are specified:

$$\text{AllValuesFrom: } C \sqsubseteq \forall P.C_E \rightsquigarrow \textbf{when } {}^{+}C(x)_d \textbf{ if } P(x,y)_{e\vee i\vee d} \textbf{ then } \neg C_E(y)_i$$
$$\textbf{when } {}^{+}P(x,y)_d \textbf{ if } C(x)_{e\vee i\vee d} \textbf{ then } \neg C_E(y)_i$$

When $C(x)$ is updated to be in state d, the trigger on C will check if $P(x,y)$ is in state e, i or d, and if so the trigger will attempt a reasoner delete $\neg C_E(y)_i$. This reasoner delete might cause the labelling to cascade to $C_E(y)$, which depends on the state of $C_E(y)$. Similarly, the trigger P will conduct a reasoner delete $\neg C_E(y)_i$ after $P(x,y)$ is labelled in state d, if $C(x)$ is in state e, i or d. With regard to the check phase, the inference rule of the table C_E is:

$$C_{E_{Int}}(y) \text{ :- } C(x)_{e\vee i}, P(x,y)_{e\vee i}$$

which selects the y appearing in non-labelled $\langle x, y\rangle$ of P where x is non-labelled in C.

IntersectionOf: as we have detailed in Section 4.5.2, OWL 2 RL supports the IntersectionOf axiom $C \equiv C_{E_1} \sqcap \ldots \sqcap C_{E_n}$ in which each of $C_{E_1} \ldots C_{E_n}$ must meet the profile restrictions. This axiom is classified by a tableaux-based reasoner into $n+1$ SubClassOf axioms (e.g. (1.1) is classified into axioms (1.5) – (1.7)):

$$\overbrace{C \sqsubseteq C_{E_1} \ldots C \sqsubseteq C_{E_n}}^{\text{n subsumption axioms}}$$

$$\underbrace{C_{E_1} \sqcap \ldots \sqcap C_{E_n} \sqsubseteq C}_{\text{1 subsumption axiom}}$$

where the first n SubClassOf axioms (i.e. $C \sqsubseteq C_{E_j}, 1 \leq j \leq n$) result in the following triggers:

$$C \sqsubseteq C_{E_j}\ 1 \leq j \leq n \rightsquigarrow \textbf{when } {}^{+}C(x)_{e \vee i} \textbf{ then } C_{E_j}(x)_i$$
$$\textbf{when } {}^{+}C(x)_{d} \textbf{ then } \neg C_{E_j}(x)_i$$

Triggers on C will attempt reasoner inserts of x to all of the tables C_{E_j} $(1 \leq j \leq n)$, after x has been inserted into C, and will attempt reasoner deletes of x from C_{E_j} $(1 \leq j \leq n)$, after $C(x)$ is updated to be in state d. Thus, transforming axioms (1.5) and (1.6), which are classified from (1.1), simply result in the following triggers:

$$\textbf{when } {}^{+}\mathsf{CzechLager}(x)_{e \vee i} \textbf{ then } \mathsf{Lager}(x)_i, \mathsf{CzechBeer}(x)_i$$
$$\textbf{when } {}^{+}\mathsf{CzechLager}(x)_{d} \textbf{ then } \neg\mathsf{Lager}(x)_i, \neg\mathsf{CzechBeer}(x)_i$$

For checking, the inference rule for each C_{E_j} $(1 \leq j \leq n)$ will include non-labelled x in C:

$$C_{E_{j\,Int}}(x) \text{ :- } C(x)_{e \vee i}\ (1 \leq j \leq n)$$

Take (1.5) and (1.6) as examples again, the inference rules for tables Lager and CzechBeer include non-labelled data in CzechLager:

$$\mathsf{Lager}_{Int}(x) \text{ :- } \mathsf{CzechLager}(x)_{e \vee i} \qquad \mathsf{CzechBeer}_{Int}(x) \text{ :- } \mathsf{CzechLager}(x)_{e \vee i}$$

For the last subsumption axiom $C_{E_1} \sqcap \ldots \sqcap C_{E_n} \sqsubseteq C$, after a data item x is inserted into one of C_{E_j} $(1 \leq j \leq n)$, a reasoner insert of x to the table C can only be attempted, if x is present in every other C_{E_k} $(k \neq j)$ in state e or i. Also, after x is labelled in one of C_{E_j} $(1 \leq j \leq n)$, a reasoner delete of x from C can only be conducted, if x is in state e, i or d in every other C_{E_k} $(k \neq j)$. Thus, the last subsumption axiom results in the following triggers:

$C_{E_1} \sqcap \ldots \sqcap C_{E_n} \sqsubseteq C \rightsquigarrow$ **when** ${}^{+}C_{E_j}(x)_{e \vee i}$ $1 \leq j \leq n$

if $C_{E_1}(x)_{e \vee i}, \ldots, C_{E_{j-1}}(x)_{e \vee i}, C_{E_{j+1}}(x)_{e \vee i}, \ldots, C_{E_n}(x)_{e \vee i}$ **then** $C(x)_i$

when ${}^{+}C_{E_j}(x)_d$ $1 \leq j \leq n$

if $C_{E_1}(x)_{e \vee i \vee d}, \ldots, C_{E_{j-1}}(x)_{e \vee i \vee d}, C_{E_{j+1}}(x)_{e \vee i \vee d}, \ldots, C_{E_n}(x)_{e \vee i \vee d}$

then $\neg C(x)_i$

Therefore, transforming the axiom (1.7) gives us the following triggers for attempting reasoner inserts and reasoner deletes:

when ${}^{+}\mathsf{Lager}(x)_{e \vee i}$ **if** $\mathsf{CzechBeer}(x)_{e \vee i}$ **then** $\mathsf{CzechLager}(x)_i$

when ${}^{+}\mathsf{CzechBeer}(x)_{e \vee i}$ **if** $\mathsf{Lager}(x)_{e \vee i}$ **then** $\mathsf{CzechLager}(x)_i$

when ${}^{+}\mathsf{Lager}(x)_d$ **if** $\mathsf{CzechBeer}(x)_{e \vee i \vee d}$ **then** $\neg\mathsf{CzechLager}(x)_i$

when ${}^{+}\mathsf{CzechBeer}(x)_d$ **if** $\mathsf{Lager}(x)_{e \vee i \vee d}$ **then** $\neg\mathsf{CzechLager}(x)_i$

By considering the last subsumption axiom, the inference rule of the table C should include all common data items in state e or i in all of C_{E_j} $(1 \leq j \leq n)$; therefore, the inference rule is demonstrated as:

$$C_{Int}(x) \text{ :- } C_{E_1}(x)_{e \vee i}, \ldots, C_{E_n}(x)_{e \vee i}$$

Consequently, from the axiom (1.7), the inference rule of the table CzechLager below includes non-labelled individuals in both tables CzechBeer and Lager.

$$\mathsf{CzechLager}_{Int}(x) \text{ :- } \mathsf{CzechBeer}(x)_{e \vee i}, \mathsf{Lager}(x)_{e \vee i}$$

Indeed, as we illustrated in the motivating example in Section 4.2, handling deletes over IntersectionOf axioms is a difficult problem, especially when deleting an individual x from C in the axiom $C \equiv C_{E_1} \sqcap \ldots \sqcap C_{E_n}$, as argued in [Dat00], which may result in a view update problem: how do we determine which C_{E_j} should the x be deleted from? SQOWL2 overcomes this, as in the label process we label every x in state i to state d in all $C_{E_1} \ldots C_{E_n}$, and afterwards during the check phase, all such labelled data items will be checked as to whether they are still inferable or not.

For the last constructor which we select for explanation, OneOf $\{a_1 \ldots a_n\}$, we first create an anonymous table (named by concatenating the names of enumerated individuals, and we denote

the table by C_{ac}) containing exact individuals defined in $\{a_1 \dots a_n\}$ as data items in state e (i.e. $C_{ac} = \{a_{1_e} \dots a_{n_e}\}$). Moreover, triggers on C_{ac} for handling inserts and deletes are specified as:

OneOf: $\{a_1 \dots a_n\} \rightsquigarrow$ **when** $^{-}C_{ac}(x)_e$ **then** rollback

when $^{-}C_{ac}(x)_i$ **if** $x \notin \{a_1 \dots a_n\}$ **then** rollback **else** ignore

when $^{-}\neg C_{ac}(x)_e$ **if** $x \in \{a_1 \dots a_n\}$ **then** rollback **else** ignore

when $^{-}\neg C_{ac}(x)_i$ **then** ignore

As can be seen, attempting an ontology insert of $C_{ac}(x)_e$ is always rolled back, no matter whether x is in $\{a_1 \dots a_n\}$ or not. Indeed, if x is equal to one of $\{a_1 \dots a_n\}$, an ontology insert of $C_{ac}(x)_e$ should be rolled back, as $C(a_1)_e \dots C(a_n)_e$ are all true. If x is not one of $\{a_1 \dots a_n\}$, an ontology insert of $C_{ac}(x)_e$ should also be rolled back, as this insert violates the OneOf axiom $\{a_1 \dots a_n\}$. For a reasoner insert of $C_{ac}(x)_i$, if x is not one of $\{a_1 \dots a_n\}$ the reasoner insert should also be rolled back; otherwise, the reasoner insert will be ignored to avoid duplicated inference. With regard to deletes, we will roll back any ontology delete to $C(a_1)_e \dots C(a_n)_e$ unless the OneOf axiom $\{a_1 \dots a_n\}$ is removed from the T-Box (however, updates to T-Box are beyond the scope of this thesis); otherwise, an ontology delete to data not in $C(a_1)_e \dots C(a_n)_e$ will be ignored to match the normal database semantics. Moreover, we ignore any reasoner delete to $C(a_1)_e \dots C(a_n)_e$ as reasoner deletes will not affect data in state e.

5.5.3 The Semantics of Property Axioms

This section continues with the definition of triggers and inference rules based on property-related axioms. We again only choose several typical constructors to illustrate the refined triggers which take data state into account, and the mapping of the other constructors into triggers and inference rules is summarised in Table 5.2.

PropertyDomain and PropertyRange: an atomic property may explicitly restrict its subject or object values to individuals from a particular OWL class. For instance, axioms (2.41) and (2.42) respectively define the PropertyDomain and PropertyRange of isFermentedBy to classes Beer and Yeast. Therefore, after an ontology or reasoner insert of $\langle x, y \rangle$ into the table isFermentedBy (representing the property isFermentedBy), one reasoner insert of x should be attempted to

Table 5.2: Triggers and inference rules generated from property axioms

DL Syntax	Triggers	Inference Rules
$P \equiv Q^-$	**when** $^+P(x,y)_{e\vee i}$ **then** $Q(y,x)_i$ **when** $^+Q(x,y)_{e\vee i}$ **then** $P(y,x)_i$ **when** $^+P(x,y)_d$ **then** $\neg Q(y,x)_i$ **when** $^+Q(x,y)_d$ **then** $\neg P(y,x)_i$	$Q_{Int}(y,x)$:- $P(x,y)_{e\vee i}$ $P_{Int}(y,x)$:- $Q(x,y)_{e\vee i}$
DisPro(P,Q)	**when** $^-P(x,y)_{e\vee i}$ **if** $Q(x,y)_{e\vee i}$ **then** rollback **when** $^-Q(x,y)_{e\vee i}$ **if** $P(x,y)_{e\vee i}$ **then** rollback	–
$P \equiv P^-$	**when** $^+P(x,y)_{e\vee i}$ **then** $P(y,x)_i$ **when** $^+P(x,y)_d$ **then** $\neg P(y,x)_i$	$P_{Int}(y,x)$:- $P(x,y)_{e\vee i}$
DisPro(P,P^-)	**when** $^-P(x,y)_{e\vee i}$ **if** $P(y,x)_{e\vee i}$ **then** rollback	–
$\top \sqsubseteq \neg\exists P$.Self	**when** $^-P(x,x)_{e\vee i}$ **then** rollback	–
$\neg P(x,y)$	**when** $^-P(x,y)_{e\vee i}$ **then** rollback	–

Beer, and another reasoner insert of y should be conducted to Yeast. This is achieved by the following trigger (note that we merge the trigger for (2.41) and the trigger for (2.42) into one single trigger, as they share the same ⟨event⟩ and ⟨condition⟩):

$$\textbf{when } {}^+\mathsf{isFermentedBy}(x,y)_{e\vee i} \textbf{ then } \mathsf{Beer}(x)_i, \mathsf{Yeast}(y)_i$$

Similarly, another trigger specified below is used for handling deletes, and it conducts two reasoner deletes $\neg\mathsf{Beer}(x)_i$ and $\neg\mathsf{Yeast}(y)_i$, after changing the state of $P(x,y)$ to d.

$$\textbf{when } {}^+\mathsf{isFermentedBy}(x,y)_d \textbf{ then } \neg\mathsf{Beer}(x)_i, \neg\mathsf{Yeast}(y)_i$$

Finally, the inference rules for tables Beer and Yeast should include x and y contained in non-labelled tuples $\langle x,y\rangle$ of the table isFermentedBy, respectively:

$$\mathsf{Beer}_{Int}(x) \text{ :- } \mathsf{isFermentedBy}(x,y)_{e\vee i}$$

$$\mathsf{Yeast}_{Int}(y) \text{ :- } \mathsf{isFermentedBy}(x,y)_{e\vee i}$$

In general, we summarise below the transformation rule for generating triggers and inference rules for PropertyDomain:

$$\begin{aligned} \text{PropertyDomain: } \top \sqsubseteq \forall P^-.C_{E_1} \rightsquigarrow \quad & \textbf{when } {}^+P(x,y)_{e\vee i} \textbf{ then } C_{E_1}(x)_i \\ & \textbf{when } {}^+P(x,y)_d \textbf{ then } \neg C_{E_1}(x)_i \\ & C_{E_{1\,Int}}(x) \text{ :- } P(x,y)_{e\vee i} \end{aligned}$$

and for PropertyRange as follows:

$$\text{PropertyRange: } \top \sqsubseteq \forall P.C_{E_2} \rightsquigarrow \quad \textbf{when } {}^{+}P(x,y)_{e\vee i} \textbf{ then } C_{E_2}(y)_i$$
$$\textbf{when } {}^{+}P(x,y)_{d} \textbf{ then } \neg C_{E_2}(y)_i$$
$$C_{E_{2\,Int}}(y) \text{ :- } P(x,y)_{e\vee i}$$

SubPropertyOf: specifies subsumption relations between properties, and a SubPropertyOf axiom $P \sqsubseteq Q$ generates two triggers on P and one inference rule for Q:

$$\text{SubPropertyOf: } P \sqsubseteq Q \rightsquigarrow \quad \textbf{when } {}^{+}P(x,y)_{e\vee i} \textbf{ then } Q(x,y)_i$$
$$\textbf{when } {}^{+}P(x,y)_{d} \textbf{ then } \neg Q(x,y)_i$$
$$Q_{Int}(x,y) \text{ :- } P(x,y)_{e\vee i}$$

As can be seen, one of the triggers on P attempts reasoner inserts to Q (i.e. the super-property table) after inserting (either ontology or reasoner inserts) data into P (i.e. the sub-property table); the other one attempts reasoner deletes over Q, if some data items in P are labelled. In addition, we specify the inference rule associated with Q to include non-labelled $\langle x, y\rangle$ in P.

TransitiveProperty: a transitive property, such as locatedIn defined by the axiom (2.54), will have $\langle x, z\rangle$ as its instance, if (x, y) and (y, z) exist in this property. Thus, we define the following transformation rule to generate triggers and the inference rule for the semantics of transitivity.

$$\text{TransitiveProperty: } P \circ P \sqsubseteq P \rightsquigarrow \quad \textbf{when } {}^{+}P(x,y)_{e\vee i} \textbf{ if } P(y,z)_{e\vee i} \textbf{ then } P(x,z)_i$$
$$\textbf{when } {}^{+}P(y,z)_{e\vee i} \textbf{ if } P(x,y)_{e\vee i} \textbf{ then } P(x,z)_i$$
$$\textbf{when } {}^{+}P(x,y)_{d} \textbf{ if } P(y,z)_{e\vee i\vee d} \textbf{ then } \neg P(x,z)_i$$
$$\textbf{when } {}^{+}P(y,z)_{d} \textbf{ if } P(x,y)_{e\vee i\vee d} \textbf{ then } \neg P(x,z)_i$$
$$P_{Int}(x,z) \text{ :- } P(x,y)_{e\vee i}, P(y,z)_{e\vee i}$$

The triggers ensure that: 1) after each ontology or reasoner insert to a transitive property table P, the triggers will compute new transitive tuples, and attempt reasoner inserts of the new tuples to the table P; 2) a tuple will be changed to be in state d if other tuples which derive it because of transitivity are labelled for deletion. Finally, for a TransitiveProperty P, the inference rule can be defined as $P_{Int}(x,z)$:- $P(x,y)_{e\vee i}, P(y,z)_{e\vee i}$.

PropertyChain: OWL 2 RL allows the specification of a property as a super property of a PropertyChain concatenating two or more properties. Here, we discuss handling the most general

situation $P_1 \circ \ldots \circ P_n \sqsubseteq P$, and triggers for handling inserts to $P_1 \ldots P_n$ are refined as:

$$
\begin{aligned}
P_1 \circ \ldots \circ P_n \sqsubseteq P \rightsquigarrow\ & \textbf{when } {}^{+}P_1(x,y)_{e\vee i} \textbf{ if } P'_{2,n}(y,z)_{e\vee i} \textbf{ then } P(x,z)_i \\
& \textbf{when } {}^{+}P_n(y,z)_{e\vee i} \textbf{ if } P'_{1,n-1}(x,y)_{e\vee i} \textbf{ then } P(x,z)_i \\
& \textbf{when } {}^{+}P_j(p,q)_{e\vee i}\ 1<j<n \textbf{ if } P'_{1,j-1}(x,p)_{e\vee i}, P'_{j+1,n}(q,z)_{e\vee i} \textbf{ then } P(x,z)_i \\
& \text{where } P'_{m,n}(x,y)_{e\vee i} = P_m(x,x_1)_{e\vee i}, P_{m+1}(x_1,x_2)_{e\vee i}, \ldots, P_n(x_{n-1},y)_{e\vee i}
\end{aligned}
$$

The first and second triggers handle ontology or reasoner inserts to the first and last subchain property, respectively. They will join the remaining subchains (i.e. $P'_{2,n}(y,z)_{e\vee i}$ and $P'_{1,n-1}(x,y)_{e\vee i}$) to derive new reasoner inserts to P. The third trigger deals with ontology or reasoner inserts to one of the middle subchains P_j (where $1<j<n$). The subchains before P_j and after P_j are respectively joined, in order to attempt reasoner inserts if new tuples in P are derived. Furthermore, the triggers for labelling are specified as:

$$
\begin{aligned}
P_1 \circ \ldots \circ P_n \sqsubseteq P \rightsquigarrow\ & \textbf{when } {}^{+}P_1(x,y)_{d} \textbf{ if } P'_{2,n}(y,z)_{e\vee i\vee d} \textbf{ then } \neg P(x,z)_i \\
& \textbf{when } {}^{+}P_n(y,z)_{d} \textbf{ if } P'_{1,n-1}(x,y)_{e\vee i\vee d} \textbf{ then } \neg P(x,z)_i \\
& \textbf{when } {}^{+}P_j(p,q)_{d}\ 1<j<n \textbf{ if } P'_{1,j-1}(x,p)_{e\vee i\vee d}, P'_{j+1,n}(q,z)_{e\vee i\vee d} \textbf{ then } \neg P(x,z)_i \\
& \text{where } P'_{m,n}(x,y)_{e\vee i\vee d} = P_m(x,x_1)_{e\vee i\vee d}, P_{m+1}(x_1,x_2)_{e\vee i\vee d}, \ldots, P_n(x_{n-1},y)_{e\vee i\vee d}
\end{aligned}
$$

Finally, for the check phase, the inference rule of the table P is specified as:

$$P_{Int}(x,y) \text{ :- } P_1(x,x_1)_{e\vee i}, P_2(x_1,x_2)_{e\vee i}, \ldots, P_n(x_{n-1},y)_{e\vee i}$$

Take the axiom (2.53) as an example, which defines the property brewedIn as a super property of brewedIn $\circ$ locatedIn, the following triggers are specified for handling ontology or reasoner inserts to brewedIn and locatedIn:

$$\textbf{when } {}^{+}\mathsf{brewedIn}(x,y)_{e\vee i} \textbf{ if } \mathsf{locatedIn}(y,z)_{e\vee i} \textbf{ then } \mathsf{brewedIn}(x,z)_i$$

$$\textbf{when } {}^{+}\mathsf{locatedIn}(y,z)_{e\vee i} \textbf{ if } \mathsf{brewedIn}(x,y)_{e\vee i} \textbf{ then } \mathsf{brewedIn}(x,z)_i$$

and the triggers defined below for the purpose of labelling:

$$\textbf{when } {}^{+}\mathsf{brewedIn}(x,y)_{d} \textbf{ if } \mathsf{locatedIn}(y,z)_{e\vee i\vee d} \textbf{ then } \neg\mathsf{brewedIn}(x,z)_i$$

$$\textbf{when } {}^{+}\mathsf{locatedIn}(y,z)_{d} \textbf{ if } \mathsf{brewedIn}(x,y)_{e\vee i\vee d} \textbf{ then } \neg\mathsf{brewedIn}(x,z)_i$$

and the following inference rule on the table brewedIn for the purpose of checking:

$$\mathsf{brewedIn}_{Int}(x,z) \text{ :- } \mathsf{brewedIn}(x,y)_{e\vee i}, \mathsf{locatedIn}(y,z)_{e\vee i}$$

Note that, in Table 5.1 and Table 5.2, we omit the axioms that do not result in the creation of triggers and inference rules.

5.6 Soundness & Completeness of SQOWL2

An inference system is sound if an ontology entails all derivations inferred by the system, and a complete inference system means that for a given ontology, the system is able to infer all possible derivations entailed by the ontology. When considering the general inference w.r.t. OWL 2 RL ontologies interpreted by the Direct Semantics, SQOWL2 is conjectured to be sound but incomplete. First, to verify the soundness of our approach, we essentially need to check for valid inputs SQOWL2 is able to compute valid inference outputs. This has already been illustrated in Section 5.5.2 and Section 5.5.3, in which all class-related and property-related semantics are axiomatically implemented as triggers.

With regard to completeness, SQOWL2 is conjectured to be a complete implementation of the OWL 2 RL/RDF rules for type inference w.r.t. OWL 2 RL interpreted by the Direct Semantics. This can be derived by using the Theorem PR1[1] in [MGH+12], which defines the requirements for an OWL 2 RL reasoner to return all and only the correct answers to certain kinds of query. Indeed, except where the UNA conflicts, all the OWL 2 RL/RDF rules are implemented as triggers in the ATIDB. All the inference tasks (i.e. type inference) and query types (i.e. conjunctive queries) which we consider exactly satisfy the requirements in Theorem PR1. Furthermore, as shown later in Chapter 7, we empirically verify the completeness conjecture of our approach by comparing SQOWL2 with a tableaux-based reasoner, and by processing the exhaustive test suites generated by SyGENiA [SGH10].

With regard to general inference (i.e. beyond type inference), our approach is incomplete, mainly because of separating the T-Box and A-Box inference. Certain T-Box subsumption relations might not be derived because of this separation, as inferring these relations would require both T-Box and A-Box statements together [Krö12b]. For instance, in an ontology containing some T-Box axioms $A \sqsubseteq \exists P.\{b\}$ and $\exists P.B \sqsubseteq C$, we cannot infer $A \sqsubseteq C$ without an

[1]The content of Theorem PR1 can be found in Appendix B.

A-Box fact $B(b)$. Nevertheless, losing this subsumption relation does not make type inference incomplete, i.e. SQOWL2 is still able to infer $C(x)$ from $A(x)$ by invoking triggers generated for $A \sqsubseteq \exists P.\{b\}$ and $\exists P.B \sqsubseteq C$ without $A \sqsubseteq C$. Furthermore, as we consider the case that the size of the T-Box is much smaller than the size of A-Boxes, it becomes possible to combine a tableaux-based reasoner and a rule-based inference engine. Sample systems that adopt this combination include DLEJena [MB10] and Minerva. As addressed in [MB08], the combination gives more complete subsumption relations than evaluating the OWL 2 RL/RDF rules.

Another difference, which we have already addressed, between our approach and other OWL reasoners is the adoption of UNA, which does not allow the case that two different knowledge objects share the same individual name. This is semantically equivalent to applying DifferentIndividuals to all individuals contained in an ontology, which implies SameIndividual, or other constructors which may derive SameIndividual axioms, such as FunctionalProperty, InverseFunctionalProperty and HasKey), cannot be used. The adoption of the UNA is not abnormal in rule-based systems, because their logical underpinning in Datalog follows the UNA and the CWA. However, SQOWL2 extends this to follow the OWA, as we have already illustrated, such as how we handle the case of $\forall P.C_E \sqsubseteq C$ with the consideration of OWA.

5.7 Summary

This chapter has described how SQOWL2 performs type inference from both inserts and deletes to the A-Box of ontologies. This is based on a variant of the DRed algorithm, which inherits the ability of handling recursive views. Such ability enables SQOWL2 to deal with all OWL 2 RL axioms, except the SameIndividual due to our adoption of UNA. We introduce four states to a data item in the ATIDB, and identify two types of database operations, and how these operations update a data item from one state to another. We have also provided an analysis about the soundness and completeness of SQOWL2. Again, the type inference is performed by invoking pre-established triggers, which incrementally analyse how the previous materialisation should be updated when updates are executed. Such inference preserves the ACID properties of transactions, and therefore satisfies transactional reasoning.

Chapter 6

Type Inference over Big Data

6.1 Introduction

In Chapter 4 and Chapter 5, we have already described how our approach performs type inference for OWL 2 ontologies the size of which is small enough to fit in an RDBMS (i.e. SQOWL2). We have focused on the problem of updating the inference materialisation, from data inserts and data deletes, in a transactional and incremental manner by using triggers. In this chapter, we introduce how our approach handles large ontologies which are too large to be stored in an RDBMS by using Spark programmes in a Big Data system (i.e. SPOWL).

We often call large-scale ontologies **Web of Data**, such as Freebase [Goo16], UniProt [Con15] and DBpedia, which have very large A-Boxes (e.g. the latest uncompressed Freebase dataset contains 1.9 billion triples and is about 250GB in size). Type inference over the sheer size of ontologies challenges the scalability of inference. Thus we move our concentration from transactional and incremental inference in an RDBMS to scalable inference for large-scale ontologies. To handle the extremely large Web of Data, many reasoners have adopted a Big Data system running a distributed computation engine, such as Hadoop and Spark, to store the Web of Data and to compute the inference results. State-of-the-art large-scale reasoners, such as WebPIE and Cichlid, usually load the explicit A-Box facts into a Big Data system first, and then compute and materialise the inference closure by evaluating a set of entailment rules, such as RDFS entailment rules [MVPG09] and OWL ter Horst rules [tH05].

To illustrate the evaluating process, we consider a fragment of the beer ontology composed of T-Box axioms (2.1) and (2.24), and A-Box facts (2.4) and (2.13), which we show again as:

$$\mathsf{PaleAle} \sqsubseteq \mathsf{Ale} \quad (2.1) \qquad \mathsf{Ale}(\mathsf{EnglishPorter}) \quad (2.4)$$

$$\mathsf{Ale} \sqsubseteq \mathsf{Beer} \quad (2.24) \qquad \mathsf{PaleAle}(\mathsf{CreamAle}) \quad (2.13)$$

Entailment rules can be expressed in the format of **if** ⟨antecedent⟩ **then** ⟨consequent⟩; for example, in the set of RDFS entailment rules, the rule used for inferring class hierarchies is:

$$\textbf{if } C_1 \sqsubseteq C_2, C_2 \sqsubseteq C_3 \textbf{ then } C_1 \sqsubseteq C_3$$

which will infer C_1 as a subclass of C_3, if C_1 is a subclass of C_2 and C_2 is a subclass of C_3. We may call this entailment rule a **schema rule**, because it outputs new T-Box axioms as the ⟨consequent⟩. Thus evaluating schema rules basically performs some T-Box inference. For instance, if we evaluate this above schema rule over (2.1) and (2.24) by matching the two T-Box axioms to the rule ⟨antecedent⟩, the T-Box axiom (2.26) will be derived:

$$\mathsf{PaleAle} \sqsubseteq \mathsf{Beer} \quad (2.26)$$

Another type of entailment rule takes some A-Box facts together with T-Box axioms as the ⟨antecedent⟩ and infers some new A-Box facts defined in the ⟨consequent⟩. We may call them **data rules**, evaluating which performs A-Box inference. For example, the following data rule will infer an individual a of C_1 as a member of C_2, if C_1 is a subclass of C_2:

$$\textbf{if } C_1 \sqsubseteq C_2, C_1(a) \textbf{ then } C_2(a)$$

Thus if we evaluate this data rule over the T-Box axiom (2.1) and the A-Box fact (2.13), we can derive a new A-Box fact that CreamAle is an instance of Ale:

$$\mathsf{Ale}(\mathsf{CreamAle}) \quad (6.1)$$

Similarly, further evaluating this data rule over (2.24) and (2.4), and over (2.24) and (6.1), gives two new A-Box facts that EnglishPorter and CreamAle are two instances of Beer:

$$\mathsf{Beer}(\mathsf{EnglishPorter}) \quad (6.2) \qquad \mathsf{Beer}(\mathsf{CreamAle}) \quad (6.3)$$

In a nutshell, the evaluating process checks as to whether there are some T-Box and A-Box statements matching the ⟨antecedent⟩ of some entailment rules; if so, statements defined in their ⟨consequent⟩ are obtained as new derivations. Entailment rules commonly adopted by

materialised approaches are RDFS entailment rules and OWL ter Horst rules; both of which cover some features of OWL 2 RL. The former contains 13 rules and can be reduced to 6 rules which are sufficient to preserve the key functionality of RDF reasoning [MVPG09]. OWL ter Horst rules consist of 24 rules, which are able to deal with a subset of more complex relations or constraints expressed in OWL 2 RL.

Large-scale reasoners (especially for those which use a materialised approach) often apply a semi-naive evaluation [AHV95], which iteratively evaluates a set of entailment rules, until no new inference can be computed. This often leads to an inefficiency issue, due to the overhead of rule matching [Krö12a]. Moreover, the evaluation usually sacrifices too much inference completeness due to the avoidance of a tableaux-based reasoner, even when the T-Box is quite simple [MB10].

This chapter presents how our approach SPOWL performs type inference for the Web of Data. For a given large ontology, SPOWL generates Spark programmes from the classified T-Box by a tableaux-based reasoner, and executes these programmes iteratively to compute and materialise the results of type inference, until no further inference can be derived.

In SPOWL, using a tableaux-based reasoner not only gives us a complete T-Box inference w.r.t. a given T-Box, but also ensures that Spark programmes are only generated for those axioms contained in the T-Box. This completely avoids evaluating entailment rules unrelated to a given large ontology. Moreover, the Spark programmes directly take relevant A-Box data as their inputs, which requires no rule matching process. We also provide an optimised order for iteratively executing the Spark programmes, in order to minimise the number of iterations until Spark programme execution can terminate (i.e. when no further inference can be derived).

SPOWL also inherits the light and fast data processing from Spark, which caches data as **Resilient Distributed Datasets** (**RDDs**) in distributed memory as much as possible on a cluster of machines, compared to Hadoop MapReduce, which requires to frequently write intermediate results to disk and thus causes an I/O overhead. In addition, Spark uses what it termed a **Directed Acyclic Graph** (**DAG**) scheduler for job scheduling, which is more flexible for parallelising jobs than Hadoop, which follows a linear-flow job scheduling of MapReduce.

Note that unlike SQOWL2 which uses triggers to perform type inference, Spark does not provide transactions, so transactional reasoning is not supported in SPOWL. Moreover, we take the assumption that the large ontologies subject to materialising inference are consistent; otherwise, the materialising process should result in a warning alongside an empty materialisation [KMR10]. Also the Spark programmes described in this chapter is rather logical, and the physical implementation details are provided in Section 7.4 of Chapter 7.

The rest of this chapter is organised as follows. Section 6.2 reviews the distributed computation engine Spark, on which the content of this chapter is based. Section 6.3 provides an overview of how SPOWL translates a classified T-Box into Spark programmes. In Section 6.4, we present the transformation rules for mapping all OWL 2 RL constructors into Spark programmes, and then we describe an optimised order for executing these Spark programmes, so that the iteration times of executing them can be minimised. Finally, Section 6.5 summarises this chapter.

6.2 Review of Spark

Spark is a distributed computation engine for fast cluster computing by using in-memory data processing. It is currently implemented in three programming languages, which are Scala, Java and Python. Spark uses its DAG scheduler to optimise job computation, and it can read/write data from/into an HDFS or other distributed file systems, such as Amazon S3[1], HBase [Geo11] and Cassandra [LM11]). Comparing Spark to other distributed computation frameworks, such as Hadoop, Pig [ORS+08] and Hive [TSJ+09], we adopt Spark due to its ability to cache data in distributed memory to eliminate I/O overhead, and Spark's more flexible and parallelised job scheduling.

6.2.1 RDD & DAG scheduler

One key functionality of Spark is the ability to cache data as RDDs in the distributed memory of a cluster of machines. Spark provides for each RDD two types of operations, which are

[1] `https://aws.amazon.com/s3/`

transformations and **actions**. Performing a transformation over an existing RDD will create a new RDD, and the existing one is called a **parent** of the new RDD. For example, the transformation filter (with a filter function) on RDD_1 will create a new RDD_2 containing filtered results of RDD_1. The following Spark code in Python[2] shows the use of filter (where lambda is a Python construct for creating anonymous functions at runtime, and as will be shown in our later examples, it is frequently used to construct functions in other Spark transformations, such as map and join):

$\mathsf{RDD}_2 = \mathsf{RDD}_1.\mathsf{filter}(\mathsf{lambda\ line} : \text{"spowl"}\ \mathsf{in\ line})$

by which RDD_2 will include the lines of RDD_1 that contain the string "spowl". Thus, RDD_1 is a parent of RDD_2. Spark creates for each RDD a DAG including all parents of the RDD, which forms a logical execution plan for generating the RDD. Thus the DAG of RDD_2 includes RDD_1.

With regard to actions, each of them executed on an RDD returns a particular value of the RDD; for example, the following code calls an action called count on RDD_2, which returns the number of lines contained in RDD_2.

$\mathsf{RDD}_2.\mathsf{count}()$

Spark adopts a **lazy evaluation** for RDD computation, which means that transformations recorded in the DAG of an RDD will not be executed until an action is called. In the above example, the filter transformation which creates RDD_2 from RDD_1 will not be executed until the count action is called on RDD_2. However, the lazy evaluation might be inefficient, if some action is frequently called over an RDD which has a complex DAG, because the RDD will be re-computed from the beginning of its DAG every time when calling the action. To overcome this, such an RDD can be **cached** in memory by Spark actions cache or persist, so that the RDD will be only computed for the first time an action is called, and afterwards, Spark will use the cached RDD from memory without unnecessary re-computation.

[2]In this thesis, we use the syntax of Python to show Spark programmes.

6.2.2 Spark vs. Hadoop

Hadoop is an open-source framework providing distributed storage and processing for large datasets on clusters of commodity machines. It offers an HDFS for the distributed data storage, and implements the MapReduce programming model for distributed computation called **Hadoop MapReduce**. The HDFS stores datasets into large blocks and distributes the datasets over the clusters of machines. For a computation job, the Hadoop MapReduce processes it via a **Map** phase followed by a **Reduce** phase. Data is firstly loaded by several mappers, where some intermediate key-value pairs are generated based on specified map functions. The intermediate pairs are then shuffled to different reducers (based on the keys), where the final result is calculated based on defined reduce functions.

The first difference of Spark from Hadoop MapReduce is that Spark caches data in distributed memory as much as possible, unlike Hadoop that needs to write/read intermediate results to/from disk and thus often suffers from an I/O overhead. The second difference is about the job scheduling between Hadoop and Spark which is illustrated in Figure 6.1.

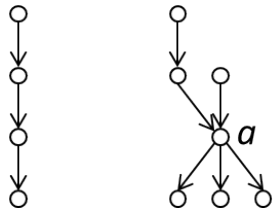

Figure 6.1: Job Scheduling between Hadoop (left) and Spark (right)

As can be seen, Hadoop schedules jobs as a linear flow, by which jobs should be executed sequentially (although each job can still be parallelised). However, the DAG scheduling provided by Spark plans jobs in a more flexible and parallelised manner (i.e. independent jobs in DAG scheduling can be computed in parallel). Moreover, if there are multiple jobs depending on the same job (e.g. the job a in Figure 6.1), the result of this job can be cached in memory by Spark, so that it can be used repeatedly without re-computation.

6.3 SPOWL Overview

The section briefly outlines SPOWL, which has a very similar architecture to SQOWL2. We also illustrate how Spark programmes are generated from a classified T-Box and then applied to the loaded data, so that the results of type inference can be computed and materialised.

6.3.1 SPOWL Architecture

Figure 6.2 shows the architecture of SPOWL, which performs type inference over large ontologies in three steps:

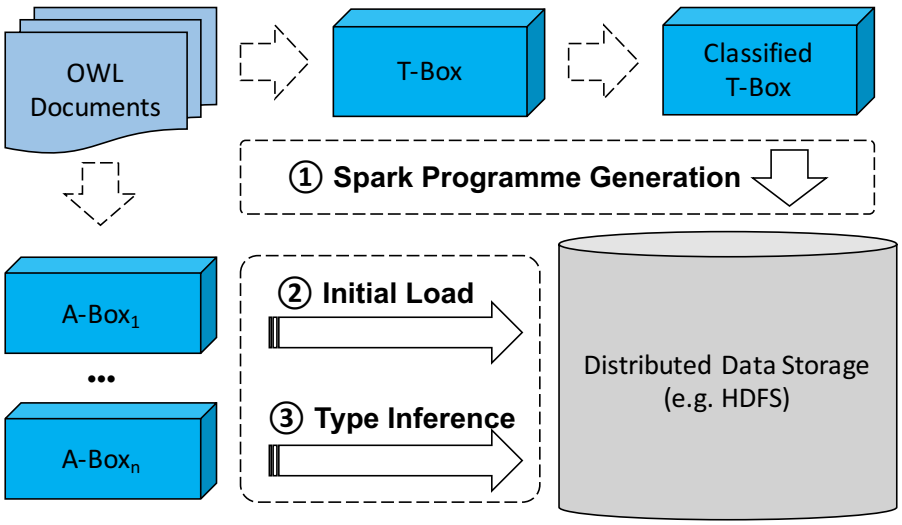

Figure 6.2: Approach Architecture for Type Inference in a Big Data System

1. After performing the classification by some tableaux-based reasoner, SPOWL compiles the classified T-Box to Spark programmes, which can be executed later over data loaded in a distributed data storage, such as an HDFS.

2. Explicit A-Box facts of a large ontology are initially loaded in the distributed storage system. We currently only consider an HDFS for storing ontological data, but this can be extended to any other Spark supported distributed storage systems.

3. We execute the Spark programmes compiled from T-Box axioms over the loaded data, so that the results of type inference are computed and persisted in the HDFS.

6.3.2 Approach Demonstration

In order to demonstrate the above three steps, we consider the following fragment of the beer ontology:

$$\mathsf{PaleAle} \sqsubseteq \mathsf{Ale} \quad (2.1) \qquad \mathsf{hasFlavour} \equiv \mathsf{hasTaste} \quad (2.49)$$

$$\mathsf{Ale} \sqsubseteq \mathsf{Beer} \quad (2.24) \qquad \mathsf{Ale}(\mathsf{EnglishPorter}) \quad (2.4)$$

$$\mathsf{Beer} \equiv \mathsf{Ale} \sqcup \mathsf{Lager} \quad (2.31) \qquad \mathsf{PaleAle}(\mathsf{CreamAle}) \quad (2.13)$$

$$\mathsf{Dom}(\mathsf{hasFlavour}, \mathsf{Ale} \sqcup \mathsf{Lager}) \quad (2.45) \qquad \mathsf{hasTaste}(\mathsf{BalticPorter}, \mathsf{Malty}) \quad (6.4)$$

Except (6.4), which is a new A-Box fact, others have been specified previously. For the five T-Box axioms, a tableaux-based reasoner classifies them as some additional axioms shown as follows (we omit the classified axioms which are irrelevant to type inference of the three A-Box facts (2.4), (2.13) and (6.4)):

$$\mathsf{PaleAle} \sqsubseteq \mathsf{Beer} \quad (2.26) \qquad \mathsf{hasTaste} \sqsubseteq \mathsf{hasFlavour} \quad (6.6)$$

$$\mathsf{Dom}(\mathsf{hasFlavour}, \mathsf{Beer}) \quad (6.5) \qquad \mathsf{hasFlavour} \sqsubseteq \mathsf{hasTaste} \quad (6.7)$$

where (2.26), classified from (2.1) and (2.24), is a SubClassOf axiom from PaleAle to Beer, axiom (6.5), obtained from (2.31) and (2.45), specifies the PropertyDomain as Beer, and finally axioms (6.6) and (6.7) are two new SubPropertyOf axioms obtained from (2.49).

We may start the generation of Spark programmes from (6.6). Assume the two properties are represented by two RDDs $\mathsf{hasTaste}_{rdd}$ and $\mathsf{hasFlavour}_{rdd}$, and after loading the A-Box fact (6.4), $\mathsf{hasTaste}_{rdd}$ contains $(\mathsf{BalticPorter}, \mathsf{Malty})$ and $\mathsf{hasFlavour}_{rdd}$ is an empty RDD. The SubPropertyOf axiom (6.6) is transformed into the following Spark programmes:

$$\mathsf{hasFlavour}_{rdd} = \mathsf{hasFlavour}_{rdd}.\mathsf{union}(\mathsf{hasTaste}_{rdd})$$

by which a union transformation on $\mathsf{hasFlavour}_{rdd}$ is performed to merge $\mathsf{hasFlavour}_{rdd}$ with $(\mathsf{BalticPorter}, \mathsf{Malty})$ contained in $\mathsf{hasTaste}_{rdd}$.

The PropertyDomain axiom (6.5) denotes that subjects of pairs in $\mathsf{hasFlavour}_{rdd}$ should be inferred as data items in Beer_{rdd} which represents the class Beer. This can be performed by:

$$\mathsf{Beer}_{rdd} = \mathsf{Beer}_{rdd}.\mathsf{union}(\mathsf{hasFlavour}_{rdd}.\mathsf{map}(\mathsf{lambda}\ (\mathsf{subject}, \mathsf{object}) : \mathsf{subject}))$$

where a map transformation is first applied on $\mathsf{hasFlavour}_{rdd}$ to map (subject, object) pairs in $\mathsf{hasFlavour}_{rdd}$ to a set containing only (subject), and then a union transformation is called to merge Beer_{rdd} with the selected set of (subject). Thus Beer_{rdd} will include the subject value BalticPorter of (BalticPorter, Malty) contained in $\mathsf{hasFlavour}_{rdd}$.

Similarly to SubPropertyOf axioms, for SubClassOf axioms, Spark programmes should merge the RDD representing the super class with data items contained in the RDD representing the subclass. Thus, axioms (2.1) and (2.24) are compiled to:

$\mathsf{Ale}_{rdd} = \mathsf{Ale}_{rdd}.\mathsf{union}(\mathsf{PaleAle}_{rdd})$

$\mathsf{Beer}_{rdd} = \mathsf{Beer}_{rdd}.\mathsf{union}(\mathsf{Ale}_{rdd})$

Suppose the A-Box facts (2.4) and (2.13) are already loaded into Ale_{rdd} and $\mathsf{PaleAle}_{rdd}$ representing the class Ale and PaleAle, respectively. By executing the above Spark codes, Ale_{rdd} will be merged with CreamAle contained in $\mathsf{PaleAle}_{rdd}$ alongside its explicit data EnglishPorter. Similarly, Beer_{rdd} will have EnglishPorter and CreamAle as its new data items in addition to BalticPorter.

Note that the order of executing Spark programmes should be in a **bottom-up** manner following the T-Box hierarchy, which reduces the number of iterations needed to terminate the inference materialisation. For example, if we first execute $\mathsf{Beer}_{rdd} = \mathsf{Beer}_{rdd}.\mathsf{union}(\mathsf{Ale}_{rdd})$ for (2.24) before executing $\mathsf{Ale}_{rdd} = \mathsf{Ale}_{rdd}.\mathsf{union}(\mathsf{PaleAle}_{rdd})$ for (2.1), then we need to execute $\mathsf{Beer}_{rdd} = \mathsf{Beer}_{rdd}.\mathsf{union}(\mathsf{Ale}_{rdd})$ again, to ensure that Beer_{rdd} includes the data items inferred to Ale_{rdd} from $\mathsf{PaleAle}_{rdd}$. The fragment of the beer ontology we have used in this section is fairly simple, and we will introduce in Section 6.4.3 how an optimised order of executing Spark programmes is specified from all OWL 2 RL axioms.

6.4 Materialising Inference Closure for OWL 2 RL

We now thoroughly show how all OWL 2 RL constructors are compiled to Spark programmes; in particular, the compiling for class-related axioms is illustrated in Section 6.4.1, and the compiling for property-related axioms is provided in Section 6.4.2. Finally, we describe in

Section 6.4.3 an optimised order of executing these programmes.

6.4.1 Compiling for OWL 2 RL Class-related Axioms

SPOWL represents OWL classes and properties as Spark RDDs (i.e. $C \rightsquigarrow C_{rdd}$ and $P \rightsquigarrow P_{rdd}$). C_{rdd} contains data items in the form of (id), while P_{rdd} contains pairs in the form of $(domain, range)$ (which Spark treats as key-value pairs). At the stage of initial loading, the A-Box facts are split into different class and property RDDs, which contain explicit facts asserted to these classes and properties, respectively. These RDDs will then have data added to them to include the implicit data derived from some OWL 2 RL axioms.

SubClassOf **and** EquivalentClasses**:**

We discuss first the case of using SubClassOf to construct subsumption relations between atomic classes (i.e. $C \sqsubseteq D$), and the subsumption between complex expressions (i.e. $C_{E_1} \sqsubseteq C_{E_2}$) are discussed when we move to other constructors. Compiling $C \sqsubseteq D$, as we have already discussed in Section 6.3.2, should ensure D_{rdd} represents the super class D and includes data items contained in C_{rdd} representing the subclass C. The Spark programme executes the union transformation to merge D_{rdd} with C_{rdd}, so that D_{rdd} will not only have its already contained data items but also data in C_{rdd}.

$$\text{SubClassOf: } C \sqsubseteq D \rightsquigarrow D_{rdd} = D_{rdd}.\mathsf{union}(C_{rdd})$$

As an EquivalentClasses axiom $C \equiv D$ is logically equivalent to two SubClassOf axioms $C \sqsubseteq D$ and $D \sqsubseteq C$, the transformation rule for compiling $C \equiv D$ into Spark programmes is intuitively:

$$\text{EquivalentClasses: } C \equiv D \rightsquigarrow D_{rdd} = D_{rdd}.\mathsf{union}(C_{rdd})$$
$$C_{rdd} = C_{rdd}.\mathsf{union}(D_{rdd})$$

Thus, after executing the above Spark programmes, C_{rdd} and D_{rdd} will contain exactly the same data items.

AllValuesFrom, SomeValuesFrom, HasValue and SelfRestriction:

OWL 2 RL only allows the expression formed by AllValuesFrom (i.e. $\forall P.C_E$) to appear as a super-class expression (i.e. $C \sqsubseteq \forall P.C_E$). The axiom $C \sqsubseteq \forall P.C_E$ expresses that for every x in C, if x is related to some y by P, then y should be inferred as an instance of C_E. Such an axiom is compiled into the Spark programmes:

$$\text{AllValuesFrom: } C \sqsubseteq \forall P.C_E \rightsquigarrow \; C_{E_{imp}} = C_{rdd}.\mathsf{map}(\mathsf{lambda}\ x_c : (x_c, x_c)).\mathsf{join}(P_{rdd})$$
$$.\mathsf{map}(\mathsf{lambda}\ (x_k, (x_c, y_p)) : y_p)$$
$$C_{E_{rdd}} = C_{E_{rdd}}.\mathsf{union}(C_{E_{imp}})$$

The Spark programmes first compute the set of data items y such that x is in C_{rdd} and (x, y) is in P_{rdd}. This is achieved by two map transformations and one join transformation. Firstly, data items x_c in C_{rdd} is mapped to key-value pairs (x_c, x_c) by .map(lambda $x_c : (x_c, x_c)$), which is then joined with pairs (of the form (x_p, y_p)) of P_{rdd} by .join(P_{rdd}). This join, called on key-value pairs of (x_c, x_c) and (x_p, y_p), will look for the case that $x_c = x_p$ and return a dataset of $(x_k, (x_c, y_p))$ pairs with all pairs of elements for each key. The $(x_k, (x_c, y_p))$ pairs are then mapped to a set containing only y_p by .map(lambda $(x_k, (x_c, y_p)) : y_p$). The set of y_p is what we need to compute, and should be added into $C_{E_{rdd}}$ by a union transformation. Note that we store in $C_{E_{imp}}$ the set of y_p which needs to be inferred to $C_{E_{rdd}}$ for a neat illustration of Spark programmes; however Spark does support the nested usage, which means the above programmes can be written as:

$$C_{E_{rdd}} = C_{E_{rdd}}.\mathsf{union}(C_{rdd}.\mathsf{map}(\mathsf{lambda}\ x_c : (x_c, x_c)).\mathsf{join}(P_{rdd}).\mathsf{map}(\mathsf{lambda}\ (x_k, (x_c, y_p)) : y_p))$$

To further explain the Spark programmes compiled from AllValuesFrom, we consider an axiom:

$$\mathsf{BritishBeer} \sqsubseteq \forall \mathsf{brewedIn}.\mathsf{BritishPlaces} \tag{6.8}$$

which specifies that all British beers are brewed in a place in Britain, and some A-Box facts:

$$\mathsf{BritishBeer}(\mathsf{EnglishIPA}) \tag{6.9}$$
$$\mathsf{BritishBeer}(\mathsf{ScottishExport}) \tag{6.10}$$
$$\mathsf{BritishBeer}(\mathsf{IrishStout}) \tag{6.11}$$
$$\mathsf{brewedIn}(\mathsf{EnglishIPA}, \mathsf{England}) \tag{6.12}$$
$$\mathsf{brewedIn}(\mathsf{ScottishExport}, \mathsf{Scotland}) \tag{6.13}$$
$$\mathsf{brewedIn}(\mathsf{MunichHelles}, \mathsf{Munich}) \tag{6.14}$$

where (6.9) – (6.11) defines EnglishIPA, ScottishExport and IrishStout as British beers, (6.12) specifies EnglishIPA is brewed in England, (6.13) denotes ScottishExport is brewed in Scotland,

and (6.14) expresses MunichHelles is brewed in Munich.

Suppose BritishBeer, brewedIn and BritishPlaces are respectively transformed into three RDDs $\mathsf{BritishBeer}_{rdd}$, $\mathsf{brewedIn}_{rdd}$ and $\mathsf{BritishPlaces}_{rdd}$, and after loading A-Box facts (6.9) – (6.14) into them, the three RDDs should be:

$$\mathsf{BritishBeer}_{rdd} = \{\mathsf{EnglishIPA}, \mathsf{ScottishExport}, \mathsf{IrishStout}\}$$

$$\mathsf{brewedIn}_{rdd} = \{(\mathsf{EnglishIPA}, \mathsf{England}), (\mathsf{ScottishExport}, \mathsf{Scotland}), (\mathsf{MunichHelles}, \mathsf{Munich})\}$$

$$\mathsf{BritishPlaces}_{rdd} = \{\}$$

By applying the transformation rule for $C \sqsubseteq \forall P.C_E$, the T-Box axiom (6.8) is compiled into:

$$\mathsf{BritishPlaces}_{imp} = \mathsf{BritishBeer}_{rdd}.\mathsf{map}(\mathsf{lambda}\ x_c : (x_c, x_c)).\mathsf{join}(\mathsf{brewedIn}_{rdd})$$
$$.\mathsf{map}(\mathsf{lambda}\ (x_k, (x_c, y_p)) : y_p)$$

$$\mathsf{BritishPlaces}_{rdd} = \mathsf{BritishPlaces}_{rdd}.\mathsf{union}(\mathsf{BritishPlaces}_{imp})$$

which we may explain step by step. First, the map transformation called on $\mathsf{BritishBeer}_{rdd}$ will map each data item x_c to a key-value pair (x_c, x_c):

$$\mathsf{BritishBeer}_{rdd}.\mathsf{map}(\mathsf{lambda}\ x_c : (x_c, x_c)) =$$
$$\{(\mathsf{EnglishIPA}, \mathsf{EnglishIPA}), (\mathsf{ScottishExport}, \mathsf{ScottishExport}), (\mathsf{IrishStout}, \mathsf{IrishStout})\}$$

Then, joining $\mathsf{BritishBeer}_{rdd}.\mathsf{map}(\mathsf{lambda}\ x_c : (x_c, x_c))$ with $\mathsf{brewedIn}_{rdd}$ (containing key-value pairs (x_p, y_p)) will give us a set of $(x_k, (x_c, y_p))$ pairs:

$$\mathsf{BritishBeer}_{rdd}.\mathsf{map}(\mathsf{lambda}\ x_c : (x_c, x_c)).\mathsf{join}(\mathsf{brewedIn}_{rdd}) =$$
$$\{(\mathsf{EnglishIPA}, (\mathsf{EnglishIPA}, \mathsf{England})), (\mathsf{ScottishExport}, (\mathsf{ScottishExport}, \mathsf{Scotland}))\}$$

As can be seen, for the pair (IrishStout, IrishStout) in $\mathsf{BritishBeer}_{rdd}.\mathsf{map}(\mathsf{lambda}\ x_c : (x_c, x_c))$, the key value IrishStout does not match any key in $\mathsf{brewedIn}_{rdd}$, and similarly, MunichHelles of the pair (MunichHelles, Munich) in $\mathsf{brewedIn}_{rdd}$ does not match any key in $\mathsf{BritishBeer}_{rdd}.\mathsf{map}(\mathsf{lambda}\ x_c : (x_c, x_c))$. Therefore, the joined result will not include them.

Furthermore, applying the map transformation which selects y_p from $(x_k, (x_c, y_p))$ pairs in $\mathsf{BritishBeer}_{rdd}.\mathsf{map}(\mathsf{lambda}\ x_c : (x_c, x_c)).\mathsf{join}(\mathsf{brewedIn}_{rdd})$ gives us $\mathsf{BritishPlaces}_{imp}$ as:

$$\mathsf{BritishPlaces}_{imp} = \{\mathsf{England}, \mathsf{Scotland}\}$$

Finally, merging $\mathsf{BritishPlaces}_{rdd}$ with $\mathsf{BritishPlaces}_{imp}$ will result $\mathsf{BritishPlaces}_{rdd}$ in:

$$\mathsf{BritishPlaces}_{rdd} = \{\mathsf{England}, \mathsf{Scotland}\}$$

which means the two individuals England and Scotland are inferred as members of BritishPlaces.

With regard to the constructor SomeValuesFrom, recalling that OWL 2 RL only allows expressions formed by it to be a subclass expression (i.e. $\exists P.C_E \sqsubseteq C$), which specifies if an individual x is related to some y of C_E by P, then x should be inferred as a member of C. Thus, $\exists P.C_E \sqsubseteq C$ are compiled to Spark programmes as:

$$\begin{aligned}\text{SomeValuesFrom: } \exists P.C_E \sqsubseteq C \rightsquigarrow\ & C_{imp} = P_{rdd}.\mathsf{map}(\mathsf{lambda}\ (x_p, y_p) : (y_p, x_p)) \\ & .\mathsf{join}(C_{E_{rdd}}.\mathsf{map}(\mathsf{lambda}\ y_{ce} : (y_{ce}, y_{ce}))) \\ & .\mathsf{map}(\mathsf{lambda}\ (y_k, (x_p, y_{ce})) : x_p) \\ & C_{rdd} = C_{rdd}.\mathsf{union}(C_{imp})\end{aligned}$$

Here, for (x_p, y_p) pairs in P_{rdd}, we need to first swap the position of x_p and y_p to form key-value pairs of (y_p, x_p) by $.\mathsf{map}(\mathsf{lambda}\ (x_p, y_p) : (y_p, x_p))$, so that they can be joined with (y_{ce}, y_{ce}) pairs formed from data items y_{ce} in $C_{E_{rdd}}$ by $.\mathsf{map}(\mathsf{lambda}\ y_{ce} : (y_{ce}, y_{ce}))$. The join between them will check for $y_p = y_{ce}$ and return a set of $(y_k, (x_c, y_{ce}))$ pairs, in which we need to select the set of items x_c by $.\mathsf{map}(\mathsf{lambda}\ (y_k, (x_c, y_{ce})) : x_c)$. Finally, C_{rdd} is merged with C_{imp}, which contains the set of items x_c.

The HasValue expression $\exists P.\{a\}$ can be either set as a subclass expression (i.e. $\exists P.\{a\} \sqsubseteq C$) or a super-class expression (i.e. $C \sqsubseteq \exists P.\{a\}$) in OWL 2 RL. We specify the transformation rule below for compiling the former case into Spark programmes:

$$\begin{aligned}\text{HasValue: } \exists P.\{a\} \sqsubseteq C \rightsquigarrow\ & C_{imp} = P_{rdd}.\mathsf{filter}(\mathsf{lambda}\ (x_p, y_p) : y_p == \text{"}a\text{"}) \\ & .\mathsf{map}(\mathsf{lambda}\ (x_p, y_p) : x_p) \\ & C_{rdd} = C_{rdd}.\mathsf{union}(C_{imp})\end{aligned}$$

As $\exists P.\{a\} \sqsubseteq C$ expresses that if an individual x is related to a constant individual a by P, then x will be derived as an instance of C, the Spark programmes first call a filter transformation to select the set of (x_p, a) pairs contained in P_{rdd}, and then use a map transformation to project x_p from (x_p, a) pairs. Finally, the set of x_p should be added into C_{rdd}. For the case of setting

$\exists P.\{a\}$ as a super-class expression, the transformation rule is specified as:

$$\text{HasValue: } C \sqsubseteq \exists P.\{a\} \rightsquigarrow P_{imp} = C_{rdd}.\mathsf{map}(\mathsf{lambda}\ x_c : (x_c, a))$$
$$P_{rdd} = P_{rdd}.\mathsf{union}(P_{imp})$$

which first forms (x_c, a) pairs for every item of data x_c in C_{rdd}, and then such pairs should be included in P_{rdd}. This captures the semantics of $C \sqsubseteq \exists P.\{a\}$, which ensures that every individual x from the class C is related to a by the property P.

Finally, for using SelfRestriction to express the semantics of local reflexivity, OWL 2 RL does not restrict its position in an axiom. As can be seen from the below transformation rule for compiling $C \sqsubseteq \exists P.\mathsf{Self}$ (i.e. the SelfRestriction is used for constructing a subclass expression), we use P_{imp} to store pairs (x_c, x_c) for data items x_c in C_{rdd} representing the class C. Then, we call the union transformation to merge P_{rdd} with P_{imp}.

$$\text{SelfRestriction: } C \sqsubseteq \exists P.\mathsf{Self} \rightsquigarrow P_{imp} = C_{rdd}.\mathsf{map}(\mathsf{lambda}\ x_c : (x_c, x_c))$$
$$P_{rdd} = P_{rdd}.\mathsf{union}(P_{imp})$$

On the other hand, when using SelfRestriction to form a subclass expression (i.e. $\exists P.\mathsf{Self} \sqsubseteq C$), compiling such an axiom to Spark programmes is specified as:

$$\text{SelfRestriction: } \exists P.\mathsf{Self} \sqsubseteq C \rightsquigarrow C_{imp} = P_{rdd}.\mathsf{filter}(\mathsf{lambda}\ (x_p, y_p) : x_p == y_p)$$
$$.\mathsf{map}(\mathsf{lambda}\ (x_p, y_p) : x_p)$$
$$C_{rdd} = C_{rdd}.\mathsf{union}(C_{imp})$$

which first select the set of pairs (x_p, y_p) where x_p is equal to y_p from P_{rdd}, and then merge C_{rdd} with x_p from these pairs.

UnionOf and IntersectionOf:

In order to avoid non-deterministic inference, in OWL 2 RL the UnionOf expression can only be set as a subclass expression. For example, we can use this constructor to form a union of two atomic classes C and D, and then set this union as a subclass of another class E. The axiom $C \sqcup D \sqsubseteq E$ is compiled into the following Spark programmes:

$$\text{UnionOf: } C \sqcup D \sqsubseteq E \rightsquigarrow E_{imp} = C_{rdd}.\mathsf{union}(D_{rdd})$$
$$E_{rdd} = E_{rdd}.\mathsf{union}(E_{imp})$$

which first creates E_{imp} containing the union of data items in both C_{rdd} and D_{rdd}, which represent the classes C and D, respectively. Then, E_{rdd} which represents the class E is merged with E_{imp} by using the RDD transformation union.

Note that for handling a more general case of setting a union of arbitrary numbers of atomic or complex class expressions as a subclass of another class (i.e. $C_{E_1} \sqcup \ldots \sqcup C_{E_n} \sqsubseteq E$), Spark programmes first respectively load data from $C_{E_1} \ldots C_{E_n}$ as $C_{E_{i_{rdd}}}$ (where $1 \leq i \leq n$), then union them together to construct E_{imp}, which is finally included in E_{rdd}. Such a transformation rule is specified as (where sc is a SparkContext object used for parallelising Spark jobs):

$$\text{UnionOf: } C_{E_1} \sqcup \ldots \sqcup C_{E_n} \sqsubseteq E \rightsquigarrow E_{imp} = \mathsf{sc.union}([C_{E_{1_{rdd}}}, \ldots, C_{E_{n_{rdd}}}])$$
$$E_{rdd} = E_{rdd}.\mathsf{union}(E_{imp})$$

With regard to IntersectionOf, OWL 2 RL allows setting the expressions formed by this constructor, such as $C \sqcap D$ which denotes common individuals of classes C and D, either as a subclass expression (e.g. $C \sqcap D \sqsubseteq E$) or a super-class expression (e.g. $E \sqsubseteq C \sqcap D$). Recall that the latter is classified as SubClassOf axioms; for example, $E \sqsubseteq C \sqcap D$ is classified to $E \sqsubseteq C$ and $E \sqsubseteq D$, and compiling these SubClassOf axioms has been described already. We then focus on the case of $C \sqcap D \sqsubseteq E$, which is compiled into Spark programmes:

$$\text{IntersectionOf: } C \sqcap D \sqsubseteq E \rightsquigarrow E_{imp} = C_{rdd}.\mathsf{intersection}(D_{rdd})$$
$$E_{rdd} = E_{rdd}.\mathsf{union}(E_{imp})$$

which use E_{imp} to include the common data in both C_{rdd} and D_{rdd} (which represent classes C and D, respectively) through an RDD transformation intersection. Then, E_{imp} is added into E_{rdd}, which denotes the class E. Again, for a more general usage of IntersectionOf $C_{E_1} \sqcap \ldots \sqcap C_{E_n} \sqsubseteq E$, we specify the transformation rule as follows:

$$\text{IntersectionOf: } C_{E_1} \sqcap \ldots \sqcap C_{E_n} \sqsubseteq E \rightsquigarrow E_{imp} = C_{E_{1_{rdd}}}.\mathsf{intersection}(C_{E_{2_{rdd}}})$$
$$\ldots.\mathsf{intersection}(C_{E_{n_{rdd}}})$$
$$E_{rdd} = E_{rdd}.\mathsf{union}(E_{imp})$$

The Spark programmes will merge E_{rdd} with E_{imp} which contains common data items in $C_{E_{1_{rdd}}} \ldots C_{E_{n_{rdd}}}$ (representing $C_{E_1} \ldots C_{E_n}$, respectively).

MinCardinality:

OWL 2 RL does not allow the use of MinCardinality. However, as SPOWL handles some extra axioms beyond OWL 2 RL, we illustrate how the unqualified and qualified use of this constructor are supported by SPOWL. When an unqualified MinCardinality expression is set as a subclass of another class (i.e. $\geqslant n\, P \sqsubseteq C$), we compile it to the following Spark programmes:

$$\begin{aligned}\text{MinCardinality: } \geqslant n\, P \sqsubseteq C \rightsquigarrow C_{imp} = P_{rdd}&.\mathsf{countByKey}().\mathsf{filter}(\mathsf{lambda}\ (x_p, x_{p_{count}}) : x_{p_{count}} \geq n)\\ &.\mathsf{map}(\mathsf{lambda}\ (x_p, x_{p_{count}}) : x_p)\\ C_{rdd} = C_{rdd}&.\mathsf{union}(C_{imp})\end{aligned}$$

As can be seen, we first call a countByKey transformation on P_{rdd}, which returns a set of pairs $(x_p, x_{p_{count}})$, where $x_{p_{count}}$ is the count of each key x_p of key-value pairs (x_p, y_p) in P_{rdd}. Then we select the keys x_p from the set of $(x_p, x_{p_{count}})$ whose $x_{p_{count}}$ is greater than or equal to n (by a filter and a map). Finally, the selected set of x_p (stored in C_{imp}) is added into C_{rdd}.

When a qualified MinCardinality expression is used as a subclass of another class (i.e. $\geqslant n\, P.C_E \sqsubseteq C$), we need to first select the set of individuals x which are related by P to at least n different individuals y from C_E, and then infer the set of x as new members of C. We specify the transformation rule for compiling $\geqslant n\, P.C_E \sqsubseteq C$ to Spark programmes as:

$$\begin{aligned}\text{MinCardinality: } \geqslant n\, P.C_E \sqsubseteq C \rightsquigarrow C_{imp} = C_{E_{rdd}}&.\mathsf{map}(\mathsf{lambda}\ y_{ce} : (y_{ce}, y_{ce}))\\ &.\mathsf{join}(P_{rdd}.\mathsf{map}(\mathsf{lambda}\ (x_p, y_p) : (y_p, x_p)))\\ &.\mathsf{map}(\mathsf{lambda}\ (y_k, (y_{ce}, x_p)) : (x_p, y_k))\\ &.\mathsf{countByKey}()\\ &.\mathsf{filter}(\mathsf{lambda}\ (x_p, x_{p_{count}}) : x_{p_{count}} \geq n)\\ &.\mathsf{map}(\mathsf{lambda}\ (x_p, x_{p_{count}}) : x_p)\\ C_{rdd} = C_{rdd}&.\mathsf{union}(C_{imp})\end{aligned}$$

As can be seen, before calling the countByKey transformation, the Spark programmes first need

to perform a join between P_{rdd} and $C_{E_{rdd}}$ in order to select the set of (x_p, y_p) pairs of P_{rdd} where y_p is in $C_{E_{rdd}}$. Afterwards, the Spark programmes are similar to the handling of $\geqslant n\, P \sqsubseteq C$, which first obtain the $(x_p, x_{p_{count}})$ pairs by the countByKey transformation, and then select the keys x_p from the set of $(x_p, x_{p_{count}})$ whose $x_{p_{count}}$ is greater than or equal to n. Finally, the Spark programmes merge C_{rdd} with the selected set of x_p stored in C_{imp}.

6.4.2 Compiling for OWL 2 RL Property-related Axioms

Moving our attention from class-related axioms to property-related axioms, we again focus on the case of using them under OWL 2 RL restrictions, and specify the transformation rules which compile them into Spark programmes.

PropertyDomain and PropertyRange:

Recalling that a PropertyDomain axiom is expressed in DL as $\top \sqsubseteq \forall P^{-}.C_{E_1}$, which specifies the semantics that the subjects x in tuples $\langle x, y\rangle$ of P must be inferred as members of C_{E_1}. A transformation rule compiles this into the following Spark programmes:

$$\text{PropertyDomain: } \top \sqsubseteq \forall P^{-}.C_{E_1} \rightsquigarrow C_{E_{1_{imp}}} = P_{rdd}.\mathsf{map}(\mathsf{lambda}\ (x_p, y_p) : x_p)$$
$$C_{E_{1_{rdd}}} = C_{E_{1_{rdd}}}.\mathsf{union}(C_{E_{1_{imp}}})$$

which first use the RDD transformation map to select all x_p from (x_p, y_p) in the property RDD P_{rdd} into $C_{E_{1_{imp}}}$, which is then added into $C_{E_{1_{rdd}}}$ representing the domain class C_{E_1}.

A PropertyRange axiom $\top \sqsubseteq \forall P.C_{E_2}$ specifies that the objects y from tuples $\langle x, y\rangle$ of P should be inferred as instances of C_{E_2}. The axiom is compiled into the following Spark programmes:

$$\text{PropertyRange: } \top \sqsubseteq \forall P.C_{E_2} \rightsquigarrow C_{E_{2_{imp}}} = P_{rdd}.\mathsf{map}(\mathsf{lambda}\ (x_p, y_p) : y_p)$$
$$C_{E_{2_{rdd}}} = C_{E_{2_{rdd}}}.\mathsf{union}(C_{E_{2_{imp}}})$$

which select the objects y_p from (x_p, y_p) contained in P_{rdd} as $C_{E_{2_{imp}}}$, which is then merged into $C_{E_{2_{rdd}}}$ representing the range class C_{E_2}.

SubPropertyOf **and** EquivalentProperties**:**

Analogous to handling SubClassOf, the transformation rule for compiling SubPropertyOf between two atomic properties $P \sqsubseteq Q$ is specified as:

$$\text{SubPropertyOf: } P \sqsubseteq Q \rightsquigarrow Q_{rdd} = Q_{rdd}.\mathsf{union}(P_{rdd})$$

The Spark programmes use the union transformation to add all pairs contained in the sub-property RDD (i.e. P_{rdd}) into the super-property RDD (i.e. Q_{rdd}). Similarly to class equivalence, an equivalent relation between two properties can be classified as two subsumption relations between them, thus the transformation for compiling $P \equiv Q$ is a union of two transformation rules for compiling $P \sqsubseteq Q$ and $Q \sqsubseteq P$:

$$\begin{aligned}\text{EquivalentProperties: } P \equiv Q \rightsquigarrow Q_{rdd} &= Q_{rdd}.\mathsf{union}(P_{rdd}) \\ P_{rdd} &= P_{rdd}.\mathsf{union}(Q_{rdd})\end{aligned}$$

The Spark programmes ensure that P_{rdd} and Q_{rdd} which represent the two properties contain the same set of pairs, so that the semantics of property equivalence is realised.

SymmetricProperty **and** InverseProperty**:**

Both SymmetricProperty and InverseProperty use the InverseOf expression P^-, where tuples in P^- swap the subjects and objects of tuples in P. A SymmetricProperty axiom is formed by specifying an equivalence between a property and the property's InverseOf expression (i.e. $P \equiv P^-$), while an InverseProperty axiom specifies that a property is equivalent to another property's InverseOf expression (i.e. $P \equiv Q^-$). We first specify the following transformation rule for compiling $P \equiv P^-$:

$$\begin{aligned}\text{SymmetricProperty: } P \equiv P^- \rightsquigarrow P_{imp} &= P_{rdd}.\mathsf{map}(\mathsf{lambda}\ (x_p, y_p) : (y_p, x_p)) \\ P_{rdd} &= P_{rdd}.\mathsf{union}(P_{imp})\end{aligned}$$

The Spark programmes first map (x_p, y_p) pairs contained in P_{rdd} (representing the property P) to a set of (y_p, x_p) pairs and store them in P_{imp}. Afterwards, the symmetry semantics is achieved by merging P_{imp} as a part of P_{rdd}.

Next, the transformation rule for compiling $P \equiv Q^-$ is:

$$\text{InverseProperty: } P \equiv Q^{-} \rightsquigarrow P_{imp} = Q_{rdd}.\mathsf{map}(\mathsf{lambda}\ (x_q, y_q) : (y_q, x_q))$$
$$P_{rdd} = P_{rdd}.\mathsf{union}(P_{imp})$$
$$Q_{imp} = P_{rdd}.\mathsf{map}(\mathsf{lambda}\ (x_p, y_p) : (y_p, x_p))$$
$$Q_{rdd} = Q_{rdd}.\mathsf{union}(P_{imp})$$

The Spark programmes first create P_{imp} containing inverse pairs of Q_{rdd}, and merge P_{rdd} with P_{imp}. Secondly, another Q_{imp} containing inverse pairs from P_{rdd} is created and added into Q_{rdd}.

PropertyChain and TransitiveProperty:

Recall the new feature PropertyChain (symbolised as $\circ$) released in OWL 2, which can be used for concatenating multiple properties. For instance, $P_1 \circ P_2$ denotes a set of $\langle x, z \rangle$ such that $\langle x, y \rangle$ is in P_1 and $\langle y, z \rangle$ is in P_2. A PropertyChain can be set as a sub property of another (or even one of the properties which were used to form the concatenation). Take $P_1 \circ P_2 \sqsubseteq P$ as an example, Spark programmes which are compiled from this axiom are:

$$\text{PropertyChain: } P_1 \circ P_2 \sqsubseteq P \rightsquigarrow P_{imp} = P_{1_{rdd}}.\mathsf{map}(\mathsf{lambda}\ (x_{p_1}, y_{p_1}) : (y_{p_1}, x_{p_1})).\mathsf{join}(P_{2_{rdd}})$$
$$.\mathsf{map}(\mathsf{lambda}\ (y_k, (x_{p_1}, z_{p_2})) : (x_{p_1}, z_{p_2}))$$
$$P_{rdd} = P_{rdd}.\mathsf{union}(P_{imp})$$

In the Spark programmes, we first map pairs (x_{p_1}, y_{p_1}) in $P_{1_{rdd}}$ to a set of inverse pairs (y_{p_1}, x_{p_1}), which are joined with (y_{p_2}, z_{p_2}) pairs of $P_{2_{rdd}}$. The join between them will check for $y_{p_1} = y_{p_2}$ and generate a set of $(y_k, (x_{P_1}, z_{p_2}))$ pairs, which are mapped to (x_{p_1}, z_{p_2}) pairs stored in P_{imp}. Then, P_{rdd} should be merged with P_{imp} to include derivations from $P_1 \circ P_2$.

For a more general case that $P_1 \circ \ldots \circ P_n \sqsubseteq P$, we need to repeatedly perform a similar joining process, in order to create P_{imp}, which contains derived pairs (x, y) from joining pairs (x, x_1) of $P_{1_{rdd}}$, (x_1, x_2) of $P_{2_{rdd}}$ $\ldots$ (x_{n-1}, y) of $P_{n_{rdd}}$. Afterwards, P_{rdd} should be merged with P_{imp}. Thus the transformation rule for compiling this axiom is:

PropertyChain: $P_1 \circ \ldots \circ P_n \sqsubseteq P \rightsquigarrow$

$$
\begin{aligned}
P_{imp} = P_{1_{rdd}}&.\mathsf{map}(\mathsf{lambda}\ (x, x_1) : (x_1, x)).\mathsf{join}(P_{2_{rdd}}) \\
&.\mathsf{map}(\mathsf{lambda}\ (x_{1_k}, (x, x_2)) : (x_2, x)).\mathsf{join}(P_{3_{rdd}}) \\
&\ldots \\
&.\mathsf{map}(\mathsf{lambda}\ (x_{n-2_k}, (x, x_{n-1})) : (x_{n-1}, x)).\mathsf{join}(P_{n_{rdd}}) \\
&.\mathsf{map}(\mathsf{lambda}\ (x_{n-1_k}, (x, y)) : (x, y)) \\
P_{rdd} = P_{rdd}&.\mathsf{union}(P_{imp})
\end{aligned}
$$

Another typical usage of the PropertyChain is to characterise a property P as a TransitiveProperty by $P \circ P \sqsubseteq P$. A TransitiveProperty P will contain $\langle x, z \rangle$, if it contains both $\langle x, y \rangle$ and $\langle y, z \rangle$. Computing the transitive closure for such P has been researched by many studies, such as [DLSW99] and [PDR05]. We adopt a simple recursive-doubling method described in [LRU14], and the Spark programmes compiled from this method are:

TransitiveProperty: $P \circ P \sqsubseteq P \rightsquigarrow$

while True **do**

$$
\begin{aligned}
&P_{imp} = P_{rdd}.\mathsf{map}(\mathsf{lambda}\ (x_p, y_p) : (y_p, x_p)).\mathsf{join}(P_{rdd}) \\
&\qquad .\mathsf{map}(\mathsf{lambda}\ (y_k, (x_p, z_p)) : (x_p, z_p)) \\
&\textbf{if}\ P_{imp}.\mathsf{isEmpty}()\ \textbf{then}\ \ \mathsf{break} \\
&P_{rdd} = P_{rdd}.\mathsf{union}(P_{imp})
\end{aligned}
$$

end

As can be seen, in each iteration of the simple recursive-doubling method, a self join on P_{rdd} (which initially contains explicit tuples of P) is performed to see whether new transitive pairs (x_p, z_p) can be computed (from pairs (x_p, y_p) and (y_p, z_p)). If so, the new pairs are stored in P_{imp} (i.e. P_{imp} is not empty), which is merged into P_{rdd} at the end of this iteration, and the updated P_{rdd} will be used for the next iteration. Otherwise, if no transitive tuples can be calculated (i.e. P_{imp} is empty), the computation of transitive closure terminates.

To determine the number of iterations required for computing a transitive closure, we interpret a TransitiveProperty as a graph, where each vertex x represents an individual x, an arc from x to y, denoted as $Arc\langle x, y \rangle$, represents x is explicitly related to y by P, and a path from x to y, $Path\langle x, y \rangle$, denotes that x is explicitly or implicitly related to y by P (i.e. through one or more arcs y is reachable from x in the graph). Thus, computing the transitive closure for P can be

interpreted as the problem of computing all $Path\langle x, y\rangle$ in the graph of this property.

The number of iterations required to terminate the computation depends on the longest path in a graph of P. If the length of an arc $Arc\langle x, y\rangle$ is set as 1, the length of $Path\langle x, y\rangle$ is the number of arcs from x to y. For example, in the graph of P, if there are arcs $Arc\langle x, a\rangle$, $Arc\langle a, b\rangle$ and $Arc\langle b, y\rangle$, then y is reachable from x via a and b, and such a path $Path\langle x, y\rangle$ is of the length 3. Note that for the case that y is reachable from x by more than one path, we consider the shortest one as its length. Continuing with the example, if the graph further contains arcs $Arc\langle x, c\rangle$ and $Arc\langle c, y\rangle$, then y is also reachable from x via c, and the length $Path\langle x, y\rangle$ should be 2, which is shorter than 3. If the longest path in a graph is of length d, a simple recursive-doubling method requires $\log_2 d$ iterations at most to finish computation of the transitive closure. However, unless d of a graph is pre-known, an extra iteration (i.e. totally $\log_2 d + 1$ iterations) is necessarily required to check as to whether P_{imp} is empty.

6.4.3 Optimised Order of Executing Spark Programmes

As we have introduced in Section 6.4.1 and Section 6.4.2, OWL 2 RL axioms are individually compiled into Spark programmes by transformation rules. For a given ontology, materialising its inference closure can be achieved by iteratively executing through the Spark programmes compiled from the classified T-Box axioms, until no new inference can be derived. Thus we would expect that the materialising process terminates with the fewest iterations of executing the Spark programmes. This requires the programmes to be executed in a bottom-up manner following the T-Box hierarchy. When materialising inference closure for large ontologies on a cluster of distributed machines, even one more iteration will impact significantly on the total performance, due to overheads of scheduling, starting and terminating distributed computation jobs. Thus, in this section, we introduce how SPOWL arranges the order of executing the Spark programmes, in order to reduce the number of iterations.

We define that a transformation rule TR_1 is **higher** than another one TR_2 (or TR_2 is **lower** than TR_1), if TR_1 takes data inferred from TR_2 as its input. For example, if we have a T-Box hierarchy composed of two SubClassOf axioms $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_3$, the transformation

rule for compiling $C_1 \sqsubseteq C_2$ (denoted as $TR_{C_1 \sqsubseteq C_2}$) is lower than the transformation rule for $C_2 \sqsubseteq C_3$ (denoted as $TR_{C_2 \sqsubseteq C_3}$), as the new data inferred because of $C_1 \sqsubseteq C_2$ contribute to the inference of $C_2 \sqsubseteq C_3$. To minimise the materialising iterations, we should execute the Spark programmes generated from the lowest transformation rules to the highest one. Thus, executing Spark programmes compiled from $TR_{C_1 \sqsubseteq C_2}$ before $TR_{C_2 \sqsubseteq C_3}$ should terminate the materialisation with only one iteration; however, executing programmes from $C_2 \sqsubseteq C_3$ before $C_1 \sqsubseteq C_2$ might require two iterations.

In order to obtain an optimised execution order, we first divide the transformation rules for OWL 2 RL axioms into three groups as shown in Figure 6.3:

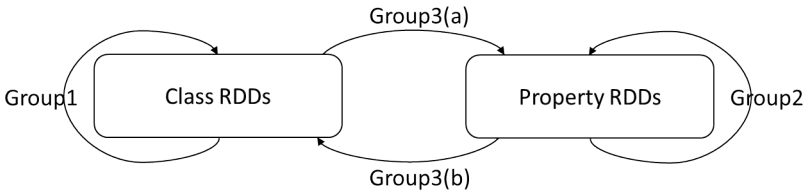

Figure 6.3: Dependence among transformation rules

1. The first group contains transformation rules which infer new data items to class RDDs from class RDDs, taking no property RDDs as input. Transformation rules fall into this group are those for compiling axioms $C \sqsubseteq D$, $C \equiv D$, $C_1 \sqcup \ldots \sqcup C_n \sqsubseteq D$ and $C_1 \sqcap \ldots \sqcap C_n \sqsubseteq D$ (where C, D and $C_1 \ldots C_n$ are all atomic classes).

2. The second group of transformation rules infer new data to property RDDs from property RDDs, taking no class RDDs as input. Such transformation rules are those for handling $P \sqsubseteq Q$, $P \equiv Q$, $P \equiv P^-$, $P \equiv Q^-$, $P \circ P \sqsubseteq P$ and $P_1 \circ \ldots \circ P_n \sqsubseteq P$ (where P, Q and $P_1 \ldots P_n$ are all atomic properties).

3. Finally, transformation rules in the third group compute new data items to class RDDs from property RDDs, or infer new data items to property RDDs from class RDDs.

   (a) Transformation rules which take some class RDDs as input (or part of the input) and generate new data to property RDDs are those for compiling $C \sqsubseteq \exists P.\{a\}$ and $C \sqsubseteq \exists P.\mathsf{Self}$.

(b) Transformation rules which compute new data to class RDDs from some property RDDs are those for handling $C \sqsubseteq \forall P.C_E$, $\exists P.C_E \sqsubseteq C$, $\exists P.\{a\} \sqsubseteq C$, $\exists P.\mathsf{Self} \sqsubseteq C$, $\geqslant n\, P \sqsubseteq C$ (and $\geqslant n\, P.D \sqsubseteq C$), $\top \sqsubseteq \forall P^{-}.C_E$ and $\top \sqsubseteq \forall P.C_E$.

Transformation rules from the first group (inferring data to class RDDs from class RDDs) are independent of the rules from the second group (inferring data to property RDDs from property RDDs). Therefore, in each of the first two groups, we can follow the class hierarchy or the property hierarchy as the bottom-up manner. We illustrate this by taking property axioms $P_1 \sqsubseteq P_2$, $P_2 \circ P_2 \sqsubseteq P_2$ and $P_2 \sqsubseteq P_3$ as an example, the property hierarchy is displayed in Figure 6.4.

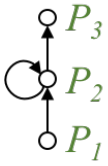

Figure 6.4: Acyclic property hierarchy

As the property hierarchy is acyclic, transformation rules should be executed as the order of $TR_{P_1 \sqsubseteq P_2}$ followed by $TR_{P_2 \circ P_2 \sqsubseteq P_2}$ followed by $TR_{P_2 \sqsubseteq P_3}$. Indeed, from $TR_{P_1 \sqsubseteq P_2}$, new data items are inferred to $P_{2_{rdd}}$ from $P_{1_{rdd}}$, and the transitive closure of $P_{2_{rdd}}$ is then computed (i.e. executing Spark programmes compiled from $TR_{P_2 \circ P_2 \sqsubseteq P_2}$); next, the transitive closure is merged into $P_{3_{rdd}}$ by using $TR_{P_2 \sqsubseteq P_3}$.

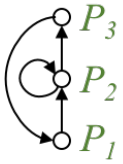

Figure 6.5: Cyclic property hierarchy

However, if we consider an extra property axiom $P_3 \equiv P_1^{-}$, the property hierarchy in Figure 6.4 becomes cyclic as shown in Figure 6.5. For a cyclic hierarchy, we cannot determine which transformation rule is the lowest; therefore, we could randomly select one of them as the first one to execute. Suppose we choose the $TR_{P_1 \sqsubseteq P_2}$ as the first transformation rule again,

after executing $TR_{P_1 \sqsubseteq P_2}$, we should then sequentially consider $TR_{P_2 \circ P_2 \sqsubseteq P_2}$ and $TR_{P_2 \sqsubseteq P_3}$. Here, due to the extra axiom $P_3 \equiv {P_1}^-$, which might infer new data to ${P_1}_{rdd}$, so $TR_{P_1 \sqsubseteq P_2}$ might need to be executed again, and so does $TR_{P_2 \circ P_2 \sqsubseteq P_2}$ and $TR_{P_2 \sqsubseteq P_3}$. In essence, the execution should terminate whenever there is no new inference to the input RDDs taken by a particular transformation rule.

Transformation rules in the third group bring the inference from class RDDs to property RDDs or vice versa, which makes it difficult for SPOWL to determine an optimised order. However, since PropertyDomain and PropertyRange axioms (inferring data from property RDDs to class RDDs) are frequently used in most ontologies, we tend to consider transformation rules in the first group are higher than those in the second group. Therefore, in general, our approach adopts an optimised order of executing transformation rules as:

1. Spark programmes compiled from the second group of transformation rules are executed to infer new data from property RDDs to property RDDs following the property hierarchy.

2. Spark programmes deriving inference from property RDDs to class RDDs are executed. Note that since transformation rules involved in this step might depend on each other, we also execute them from lower transformation rules to higher rules.

3. Spark programmes generated from the first group of transformation rules are processed to derive new data from class RDDs to class RDDs following the class hierarchy.

4. Whenever the ontology contains axioms which could derive new inference from class RDDs to property RDDs, we check whether new data items are inferred because of them; if so we re-conduct the previous three steps until the input taken by higher-level transformation rules contains no newly inferred data.

6.5 Summary

To sum up, this chapter has described how our work SPOWL compiles OWL 2 RL axioms to Spark programmes, which support type inference over large ontologies. Unlike most large

reasoners which simply evaluate a set of entailment rules for materialising the inference closure, SPOWL compiles a classified T-Box to Spark programmes, which are used for inference materialisation. Moreover, compared to reasoners using Hadoop, SPOWL benefits from Spark which uses distributed memory as much as possible, and schedules jobs in a more flexible and parallelised manner by the DAG scheduler.

In particular, we have specified transformation rules for compiling all OWL 2 RL axioms into Spark programmes, which can be divided into three groups. We have introduced an optimised order of executing the three groups of Spark programmes, which might reduce execution iterations until the computation of inference closure can terminate.

However, our approach restricts itself to consider simple and small T-Boxes, while ontologies with complex and large structure (e.g. SNOMED-CT and Gene ontology) are beyond the scope of SPOWL. Also, if dependencies among the Spark programmes contain many cycles, an optimised order of execution is difficult to obtain, but this is highly unlikely in real-world ontologies. As the Spark programmes are presented in a logical format, implementation of them to real code is provided in Chapter 7, along with some tuning strategies.

Chapter 7

Implementation & Evaluation

7.1 Introduction

In Chapters 4, 5 and 6, we have described our approach, SQOWL2 and SPOWL, in an implementation-independent way. In particular, triggers defined in SQOWL2 and Spark programmes specified in SPOWL are presented in a logical format. In this chapter, we further present the implementation details of SQOWL2 in Section 7.2 and of SPOWL in Section 7.4, which respectively describe how logical triggers are translated into physical triggers, and how logical Spark programmes are implemented into real code. From the perspective of adopting our approach into standard database applications in which the number of queries is much larger than the number of updates, we have also evaluated the two implementations, by conducting a series of experiments and analysing the experiment results. In particular, in the evaluation of SQOWL2 detailed in Section 7.3, we experimentally verify our hypothesis that, when compared to non-materialising inference, materialising inference results leads to faster query processing at the expense of slower data updating. In addition, the results show that the overhead due to inference by SQOWL2 is not impractical to apply to database applications, and is much less than one order of magnitude in size. Moreover, we empirically test the soundness and completeness of SQOWL2 over the benchmark data. Since SPOWL performs inference over much larger ontologies, the scalability of inference materialisation and query processing is instead our focus when evaluating SPOWL in Section 7.5. We conduct experiments over much larger datasets, and evaluation results show that SPOWL scaled linearly for inference materialisation

and quadratically for query processing. Finally, Section 7.6 summarises this chapter.

7.2 Implementation of SQOWL2

In this section, we review the implementation details of SQOWL2. In particular, Section 7.2.1 details a front-end and a back-end schema used for representing data state, which are essential for handling data updates especially deletes. After that, in Section 7.2.2, we thoroughly illustrate how physical triggers are created from logical triggers. Finally, in Section 7.2.3 we summarise some optimisations which we adopt to improve the performance of SQOWL2. Note that all SQL statements follow the syntax of Transact SQL, the variant of the SQL language used by Microsoft SQL Server.

7.2.1 Two Schemas to Represent Data and State

In Section 5.4.1, we have introduced a logical state (i.e. $\emptyset$, e, i and d) for each item of data stored in the ATIDB, in order to perform type inference from data updates especially deletes. To implement data state, in the canonical schema, which contains one-column class tables and two-column property tables, we can simply add an extra column st (representing the data state) to each class or property table, based on the following mapping:

$$\text{Class: } C \rightsquigarrow C(\underline{id}, st)$$

$$\text{Property: } P \rightsquigarrow P(\underline{domain, range}, st)$$

Tables can be implemented in an RDBMS by some SQL CREATE TABLE statements. For example, based on the above mapping rule from an OWL class to a table, the statement for creating the table Beer representing the OWL class Beer is:

```
CREATE TABLE Beer(
       id VARCHAR(100) NOT NULL,
       st CHAR(1) NOT NULL,
       CONSTRAINT Beer_PK PRIMARY KEY (id))
```

Note that, in the above statement, setting a key (e.g. id) is achieved by a PRIMARY KEY constraint. Another sample SQL statement which creates the table hasDescription for the OWL

property hasDescription is:

```
CREATE TABLE hasDescription(
        domain VARCHAR(100) NOT NULL,
        range VARCHAR(100) NOT NULL,
        st CHAR(1) NOT NULL,
        CONSTRAINT hasDescription_PK PRIMARY KEY (domain,range))
```

Moreover, SubClassOf axioms between atomic classes and SubPropertyOf axioms between atomic properties should be mapped as FKs. For example, because Ale is specified as a subclass of Beer by (2.24), when creating the table Ale for Ale, a FOREIGN KEY constraint implementing $\mathsf{Ale}(\mathsf{id}) \overset{\mathsf{fk}}{\Rightarrow} \mathsf{Beer}(\mathsf{id})$ should be added:

```
CREATE TABLE Ale(
        id VARCHAR(100) NOT NULL,
        st CHAR(1) NOT NULL,
        CONSTRAINT Ale_PK PRIMARY KEY (id),
        CONSTRAINT Ale_SubOf_Beer FOREIGN KEY (id) REFERENCES Beer(id))
```

Similarly, since hasColour is a sub property of hasDescription defined by (2.47), when creating the table hasColour, $\mathsf{hasColour}(\mathsf{domain}, \mathsf{range}) \overset{\mathsf{fk}}{\Rightarrow} \mathsf{hasDescription}(\mathsf{domain}, \mathsf{range})$ is added as:

```
CREATE TABLE hasColour(
        domain VARCHAR(100) NOT NULL,
        range VARCHAR(100) NOT NULL,
        st CHAR(1) NOT NULL,
        CONSTRAINT hasColour_PK PRIMARY KEY (domain,range),
        CONSTRAINT hasColour_SubOf_hasDescription FOREIGN KEY (domain,range)
                    REFERENCES hasDescription(domain,range))
```

The FK constraints capture the semantics of SubClassOf or SubPropertyOf by validating that data items contained in the subsumee are always in the subsumer. This is already achieved by triggers which attempt an insert of data items to the subsumer table after the same items of data have been inserted to the subsumee table. As we will illustrate in Section 7.3, enabling FK checking would slow down the performance of data loading and removing, but would improve the query processing. Therefore, we would disable the FK checking when SQOWL2 is subject to data inserts and deletes. and re-enable them when SQOWL2 is subject to query processing.

However, in some use cases we might only want to show users or applications the data items, but not their states, as exposing the state columns will expose the implementation details.

In order to allow this, we establish another schema, called a front-end schema alongside the canonical schema (which we treat as a back-end schema), as outlined in Figure 7.1.

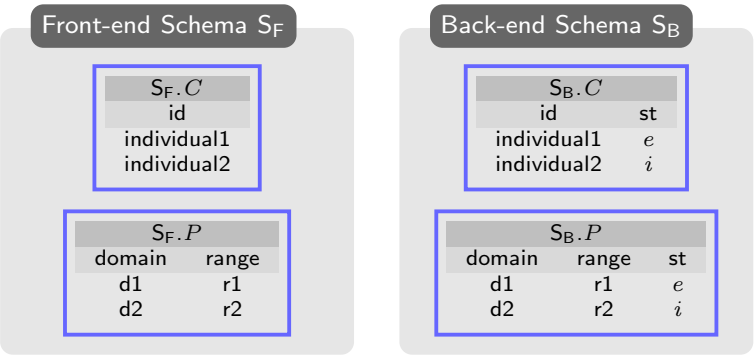

Figure 7.1: Front-end and back-end schemas

The back-end schema (which is denoted by S_B) consists of materialised tables storing both data items and their states, whereas the front-end schema (which is denoted by S_F) contains views showing users or applications only the data items stored in S_B.

We apply the standard SQL CREATE VIEW statements for creating views in S_F, which follows the mappings from OWL classes and properties to Datalog rules:

$$C \rightsquigarrow S_F.C(id) \text{ :- } S_B.C(id, st)$$

$$P \rightsquigarrow S_F.P(domain, range) \text{ :- } S_B.P(domain, range, st)$$

Therefore, the tables S_B.Beer, S_B.Ale, S_B.hasDescription and S_B.hasColour result in four views S_F.Beer, S_F.Ale, S_F.hasDescription and S_F.hasColour, which can be created by:

```
CREATE VIEW SF.Beer AS
       SELECT id FROM SB.Beer
CREATE VIEW SF.Ale AS
       SELECT id FROM SB.Ale
CREATE VIEW SF.hasDescription AS
       SELECT domain,range FROM SB.hasDescription
CREATE VIEW SF.hasColour AS
       SELECT domain,range FROM SB.hasColour
```

Note that only the front-end schema S_F is exposed to database users for showing the conceptual views of an ontology, and also for accepting ontology inserts and deletes, which are then mapped onto S_B, where the process of type inference is performed.

7.2.2 Implementing Logical Triggers as Physical Triggers

Recall that logical triggers can be divided into two types denoted by the "−" or "+" prefix in their fields of ⟨event⟩. INSTEAD OF triggers in Transact SQL (or BEFORE triggers in PL/pgSQL [Mom01]) are used to implement triggers with "−" prefixed events, and AFTER triggers are used for triggers having "+" events. The logical triggers presented in Chapter 4 and Chapter 5 are a platform-independent method of writing triggers, and are then translated by SQOWL2 into physical triggers. In order to illustrate the implementation of physical triggers from logical triggers in detail, we reconsider the T-Box axioms (2.24) (i.e. Ale $\sqsubseteq$ Beer), (2.28) (i.e. Beer $\equiv$ LiquidBread) and (5.1) (i.e. IrishBeer $\sqsubseteq$ LiquidBread) used in Section 5.2.

Implementing Triggers for Inserts

Ontology inserts, which are executed over views in $\mathsf{S_F}$, will be automatically transferred to the corresponding tables in $\mathsf{S_B}$ by the RDBMS. For example, if we attempt two ontology inserts of $\mathsf{v_1}$ and $\mathsf{v_2}$ into the view $\mathsf{S_F}$.Ale, which are displayed as the following SQL INSERT statement:

```
INSERT INTO SF.Ale(id)
VALUES ('v1'),('v2')
```

This INSERT statement will be mapped by the RDBMS into the following SQL statement:

```
INSERT INTO SB.Ale(id,st)
VALUES ('v1',null),('v2',null)
```

which is executed on $\mathsf{S_B}$.Ale (in Schema $\mathsf{S_B}$). The two inserts on $\mathsf{S_B}$.Ale are then captured by the BeforeInsert$\mathsf{S_B}$AleTrigger (detailed in Figure 7.2) created on $\mathsf{S_B}$.Ale.

The BeforeInsert$\mathsf{S_B}$AleTrigger contains three parts:

1. if the data is already present and has been explicitly stated (i.e. the data is already persisted in state e), the transaction of inserting such data will be rolled back, in order to avoid repeated ontology inserts.

2. if the data is already present and has been implicitly stated (i.e. the data is persisted in state i), then the data will be updated to explicit state, in order to give it the explicit

```
CREATE TRIGGER BeforeInsertSBAleTrigger
ON SB.Ale INSTEAD OF INSERT AS BEGIN
 -- part 1: rollback repeated ontology inserts
 IF EXISTS (SELECT *
    FROM inserted
         JOIN SB.Ale
         ON   inserted.id=SB.Ale.id
         AND  SB.Ale.st='e'
         AND  inserted.st IS NULL)
   RAISERROR('ERROR MESSAGE', 10, 1)
   ROLLBACK TRANSACTION
   RETURN
 -- part 2: update implicit data with state e
 UPDATE SB.Ale SET st='e'
 WHERE SB.Ale.id IN
     (SELECT SB.Ale.id
      FROM SB.Ale
           JOIN inserted
           ON   SB.Ale.id=inserted.id
           AND  inserted.st IS NULL
           AND  SB.Ale.st='i')
 -- part 3: persist data if it does not exists
 INSERT INTO SB.Ale(id,st)
 SELECT DISTINCT id,COALESCE(st,'e')
 FROM   inserted
 WHERE inserted.id NOT IN
     (SELECT id FROM SB.Ale)
END
```

Figure 7.2: Before insert trigger on $\mathsf{S_B}$.Ale

semantics (i.e. i is updated to e).

3. if the data is simply not present (i.e. the data is logically in state ø), an insert of the new data with the explicit state e will be performed.

Supposing that $\mathsf{v_1}$ and $\mathsf{v_2}$ are not stored in the ATIDB, the original SQL statement which inserts the two items of data is transformed into the following SQL statement:

```
INSERT INTO SB.Ale(id,st)
VALUES ('v1','e'),('v2','e')
```

which inserts $\langle \mathsf{v_1}, \mathsf{e} \rangle$ and $\langle \mathsf{v_2}, \mathsf{e} \rangle$ into $\mathsf{S_B}$.Ale, and stores $\mathsf{v_1}$ and $\mathsf{v_2}$ in state e in $\mathsf{S_B}$.Ale.

Our implementation in an RDBMS has another important feature: all triggers are implemented with table-level semantics (which treats multiple data items in bulk) rather than row-level

semantics (which treats multiple data items individually), in order to improve the performance of type inference. Take the physical trigger $\mathsf{AfterInsertS_BAleTrigger}$ (shown in Figure 7.3) on $\mathsf{S_B.Ale}$ for the logical trigger **when** ${}^{+}\mathsf{Ale}(x)_{e\vee i}$ **then** $\mathsf{Beer}(x)_i$ (specified for (2.24)) as an example:

```
CREATE TRIGGER AfterInsertSBAleTrigger
ON SB.Ale AFTER INSERT AS
BEGIN
 -- generate reasoner inserts to SB.Beer
 INSERT INTO SB.Beer(id,st)
 SELECT DISTINCT id,'i'
 FROM   inserted
END
```

Figure 7.3: After insert trigger on $\mathsf{S_B.Ale}$

If there are multiple data items inserted into $\mathsf{S_B.Ale}$, all of these items of data will be gathered as one reasoner insert into $\mathsf{S_B.Beer}$ rather than inserting each of them separately. Therefore, after storing $\mathsf{v_1}$ and $\mathsf{v_2}$ in $\mathsf{S_B.Ale}$, the $\mathsf{AfterInsertS_BAleTrigger}$ will be fired to produce another SQL statement shown as follows:

```
INSERT INTO SB.Beer(id,st)
VALUES ('v1','i'),('v2','i')
```

which inserts $\langle \mathsf{v_1}, \mathsf{i} \rangle$ and $\langle \mathsf{v_2}, \mathsf{i} \rangle$ into $\mathsf{S_B.Beer}$, so that $\mathsf{v_1}$ and $\mathsf{v_2}$ are stored in state i in $\mathsf{S_B.Beer}$. The inference process should cascade to the table $\mathsf{S_B.LiquidBread}$ because of the logical trigger **when** ${}^{+}\mathsf{Beer}(x)_{e\vee i}$ **then** $\mathsf{LiquidBread}(x)_i$ as a result of the axiom (2.28). This is conducted by the $\mathsf{AfterInsertS_BBeerTrigger}$ shown in Figure 7.4 (similar to $\mathsf{AfterInsertS_BAleTrigger}$), i.e. $\mathsf{AfterInsertS_BBeerTrigger}$ captures the insert of $\langle \mathsf{v_1}, \mathsf{i} \rangle$ and $\langle \mathsf{v_2}, \mathsf{i} \rangle$ into $\mathsf{S_B.Beer}$, and afterwards inserts the same two tuples to $\mathsf{S_B.LiquidBread}$.

```
CREATE TRIGGER AfterInsertSBBeerTrigger
ON SB.Beer AFTER INSERT AS
BEGIN
 -- generate reasoner inserts to SB.LiquidBread
 INSERT INTO SB.LiquidBread(id,st)
 SELECT DISTINCT id,'i'
 FROM   inserted
END
```

Figure 7.4: After insert trigger on $\mathsf{S_B.Beer}$

Note that after persisting $\langle \mathsf{v_1}, \mathsf{i} \rangle$ and $\langle \mathsf{v_2}, \mathsf{i} \rangle$ in $\mathsf{S_B.LiquidBread}$, the persistence will be captured by the after trigger on this table (i.e. $\mathsf{AfterInsertS_BLiquidBreadTrigger}$). As the two classes

LiquidBread and Beer are equivalent, the semantics of this equivalence results in a repeated reasoner insert of $\langle v_1, i \rangle$ and $\langle v_2, i \rangle$ to S_B.Beer as illustrated in Figure 7.5.

```
CREATE TRIGGER AfterInsertSBLiquidBreadTrigger
ON SB.LiquidBread AFTER INSERT AS
BEGIN
 -- generate reasoner inserts to SB.Beer
 INSERT INTO SB.Beer(id,st)
 SELECT DISTINCT id,'i'
 FROM   inserted
END
```

Figure 7.5: After insert trigger on S_B.LiquidBread

However, the reasoner insert is duplicated and will be ignored by a before insert trigger created on S_B.Beer (we omit the definition of this physical trigger as it is very similar to BeforeInsertS_BAleTrigger shown in Figure 7.2).

Implementing Triggers for Deletes

Moving our attention to implementing triggers for deletes, our implementation needs to guarantee that the labelling process finishes before conducting the checking process, i.e. for ontology deletes of some data items, inferred data from them must be all labelled before checking. In order to control the labelling and checking (which are processed in S_B), we create for each view in S_F a 'before delete' trigger (*a.k.a.* an INSTEAD OF DELETE trigger in Transact SQL), such as BeforeDeleteS_FAleTrigger shown in Figure 7.6 for handling ontology deletes over S_F.Ale.

Each 'before delete' trigger should consist of three essential parts:

1. the first part is used to detect inconsistent ontology deletes, which are those attempting to delete an item of data in the implicit state i. A rollback of the transaction attempting such inconsistent ontology deletes should be conducted.

2. The functionality of the second part is to trigger the labelling process, and as exemplified in the BeforeDeleteS_FAleTrigger, such a process is mapped to an SQL UPDATE statement. We show later how triggers in S_B handle the UPDATE statement, so that the labelling process cascades to all data items which depend on the ontology delete.

```
CREATE TRIGGER BeforeDeleteSFAleTrigger
ON SF.Ale INSTEAD OF DELETE AS
BEGIN
 -- part1: rollback inconsistent deletes
 IF EXISTS
       (SELECT *
        FROM deleted
             JOIN SB.Ale
             ON   deleted.id=SB.Ale.id
             AND  SB.Ale.st='i')
   RAISERROR('ERROR MESSAGE', 10, 1)
   ROLLBACK TRANSACTION
   RETURN
 -- part2: start labelling process
 UPDATE SB.Ale
 SET    st='d'
 WHERE  id IN (SELECT id FROM deleted)
 AND    st='e'
 -- part3: start checking process
 DELETE FROM SB.Ale
 WHERE st='d'
END
```

Figure 7.6: Before delete trigger on $\mathsf{S_F}$.Ale

3. The third part is used for checking, which is achieved by starting an SQL DELETE. Again, the DELETE statement will be captured by some other triggers in $\mathsf{S_B}$, so that the checking cascades to all labelled data items in the same order that these data items were labelled.

Note that the third part can only start the checking phase, when the second part is finished (i.e. the whole set of data which has an inferring logic from the data of the ontology delete is updated to state *d*). To illustrate this, we continue with the example of ontology inserting $\mathsf{v_1}$ and $\mathsf{v_2}$ to $\mathsf{S_F}$.Ale (i.e. tuples $\langle \mathsf{v_1}, \mathsf{e} \rangle$ and $\langle \mathsf{v_2}, \mathsf{e} \rangle$ are materialised in $\mathsf{S_B}$.Ale, and tuples $\langle \mathsf{v_1}, \mathsf{i} \rangle$ and $\langle \mathsf{v_2}, \mathsf{i} \rangle$ are materialised in both $\mathsf{S_F}$.Beer and $\mathsf{S_F}$.LiquidBread). Suppose we now delete $\mathsf{v_1}$ from $\mathsf{S_F}$.Ale. Such an ontology delete should be executed as an SQL DELETE statement:

```
DELETE FROM SF.Ale
WHERE id ='v1'
```

Since $\langle \mathsf{v_1}, \mathsf{e} \rangle$ is stored in $\mathsf{S_B}$.Ale, the above ontology delete is legal, but a labelling process should be conducted. Thus, part 2 of the BeforeDelete$\mathsf{S_F}$AleTrigger changes this to an SQL UPDATE:

```
UPDATE SB.Ale
SET    st='d'
WHERE  id='v1'
AND    st='e'
```

which updates $\langle \mathsf{v}_1, \mathsf{e} \rangle$ in $\mathsf{S_B}$.Ale to $\langle \mathsf{v}_1, \mathsf{d} \rangle$. This state update is then detected by an after update trigger $\mathsf{AfterUpdateS_BAleTrigger}$ on $\mathsf{S_B}$.Ale shown in Figure 7.7, which continues the labelling.

```
CREATE TRIGGER AfterUpdateSBAleTrigger
ON SB.Ale AFTER UPDATE AS
BEGIN
 -- labelling cascades to SB.Beer
  UPDATE SB.Beer
  SET    st='d'
  WHERE  st='i'
  AND    id IN (SELECT id
                FROM   inserted
                WHERE  st='d')
END
```

Figure 7.7: After update trigger on $\mathsf{S_B}$.Ale

Because v_1 is implicitly stored in the table $\mathsf{S_B}$.Beer (i.e. $\langle \mathsf{v}_1, \mathsf{i} \rangle$ is materialised in $\mathsf{S_B}$.Beer), this trigger will generate another SQL UPDATE statement:

```
UPDATE SB.Beer
SET    st='d'
WHERE  id='v1'
AND    st='i'
```

which will further update the state of v_1 in $\mathsf{S_B}$.Beer to d. Similarly, this update in $\mathsf{S_B}$.Beer will be captured by another after update trigger created on this table, which continues to label the same data v_1 in the table $\mathsf{S_B}$.LiquidBread (i.e. $\langle \mathsf{v}_1, \mathsf{d} \rangle$ is stored in $\mathsf{S_B}$.LiquidBread). At this point, the cascade of the labelling process terminates.

Because the labelling has finished, part 3 of the $\mathsf{BeforeDeleteS_FAleTrigger}$ can now launch the check process by attempting to execute an SQL DELETE statement:

```
DELETE FROM SB.Ale
WHERE st='d'
```

which in our example attempts to delete v_1 from $\mathsf{S_B}$.Ale. Note that this DELETE statement is just used for conducting the checking, but whether to delete or not still depends on a before delete trigger created on $\mathsf{S_B}$.Ale, which is the $\mathsf{BeforeDeleteS_BAleTrigger}$ shown in Figure 7.8:

```
CREATE TRIGGER BeforeDeleteSBAleTrigger
ON SB.Ale INSTEAD OF DELETE AS
BEGIN
 -- part1: keep inferable data
 -- no action required
 -- part2: remove non-inferable data
 DELETE FROM SB.Ale
 WHERE  st='d'
END
```

Figure 7.8: Before delete trigger on $\mathsf{S_B}$.Ale

This trigger essentially updates the state of data items which are still derivable to i (defined in part 1), and deletes the data which is non-inferable (specified in part 2). Since we only consider axioms (2.24), (2.28) and (5.1), the inference rule created for $\mathsf{S_B}$.Ale is empty (i.e. no action is required in part 1 of BeforeDelete$\mathsf{S_B}$AleTrigger), and part 2 of BeforeDelete$\mathsf{S_B}$AleTrigger simply delete $\mathsf{v_1}$ from $\mathsf{S_B}$.Ale, because $\mathsf{v_1}$ is in state d.

```
CREATE TRIGGER AfterDeleteSBAleTrigger
ON SB.Ale After DELETE AS
BEGIN
 -- checking cascades to SB.Beer
 DELETE FROM SB.Beer
 WHERE  SB.Beer.id IN (SELECT id
                       FROM deleted)
 AND    SB.Beer.st='d'
END
```

Figure 7.9: after delete trigger on $\mathsf{S_B}$.Ale

Analogous to labelling, the check phase should also cascade over the same data items that were labelled. Thus, we create an after delete trigger on the table $\mathsf{S_B}$.Ale (i.e. AfterDelete$\mathsf{S_B}$AleTrigger shown in Figure 7.9), which detects the deletion of $\mathsf{v_1}$, and cascades the check to the same data stored in the table $\mathsf{S_B}$.Beer by attempting an SQL DELETE statement below over $\mathsf{S_B}$.Beer:

```
DELETE FROM SB.Beer
WHERE st='d'
```

Again, a before delete trigger on $\mathsf{S_B}$.Beer named BeforeDelete$\mathsf{S_B}$BeerTrigger shown in Figure 7.10 (similar to BeforeDelete$\mathsf{S_B}$AleTrigger) handles the attempt at deleting $\mathsf{v_1}$ from $\mathsf{S_B}$.Beer.

This time, since data in $\mathsf{S_B}$.Beer might be inferred from both $\mathsf{S_B}$.Ale and $\mathsf{S_B}$.LiquidBread, part 1 of BeforeDelete$\mathsf{S_B}$BeerTrigger is not empty, and should check the data under the attempt at

```
CREATE TRIGGER BeforeDeleteSBBeerTrigger
ON SB.Beer INSTEAD OF DELETE AS
BEGIN
 -- part1: keep still inferable data
 UPDATE SB.Beer
 SET    st='i'
 WHERE  st='d'
 AND    id IN (SELECT id
               FROM SB.Ale
               WHERE st<>'d'
               UNION
               SELECT id
               FROM SB.LiquidBread
               WHERE st<>'d')
 -- part2: remove non-inferable data
 DELETE FROM SB.Beer
 WHERE  st='d'
END
```

Figure 7.10: Before delete trigger on $\mathsf{S_B}$.Beer

deleting ($\mathsf{v_1}$ in our example) can still be inferred or not. Because $\mathsf{v_1}$ is not present in $\mathsf{S_B}$.Ale, and was updated to $\langle \mathsf{v_1}, \mathsf{d} \rangle$ in $\mathsf{S_B}$.LiquidBread, it cannot be derived and should be deleted by part 2 of BeforeDelete$\mathsf{S_B}$BeerTrigger.

If we continue the checking of $\mathsf{v_1}$ in the table $\mathsf{S_B}$.LiquidBread, we find that $\mathsf{v_1}$ should be removed from $\mathsf{S_B}$.LiquidBread as $\mathsf{v_1}$ is not stored in tables $\mathsf{S_B}$.Beer nor $\mathsf{S_B}$.IrishBeer. Thus, the checking phase caused by deleting $\mathsf{v_1}$ from $\mathsf{S_F}$.Ale is finished (i.e. the third part of BeforeDelete$\mathsf{S_F}$AleTrigger terminates), and so does the whole process of handling this ontology delete (i.e. DELETE FROM SF.Ale WHERE id='v1').

7.2.3 Optimisation in SQOWL2

Indexes & Foreign Keys

Most reasoners over ontologies with large A-Boxes consider query processing as the most important inference task. As a materialised approach, queries handled by SQOWL2 only need to read the inference materialisation. In addition, we may use indexes and FKs provided by an RDBMS to further improve the performance of query processing.

Firstly, in the back-end schema $\mathsf{S_B}$, we create for every class table C an index on the column

(id), and for every property table P two indexes on $(domain, range)$ and $(range, domain)$. Indexes will benefit queries, especially when selecting specific rows from tables, or requiring results to be sorted in a particular order. Secondly, besides creating the FKs for SubClassOf and SubPropertyOf as described in Section 7.2.1, we further add FKs for PropertyDomain and PropertyRange axioms, based on the following rules:

$$\text{PropertyDomain: } \top \sqsubseteq \forall P^{-}.C_{E_1} \rightsquigarrow P(domain) \overset{\text{fk}}{\Rightarrow} C_{E_1}(id)$$

$$\text{PropertyRange: } \top \sqsubseteq \forall P.C_{E_2} \rightsquigarrow P(range) \overset{\text{fk}}{\Rightarrow} C_{E_2}(id)$$

FKs representing PropertyDomain and PropertyRange axioms are especially useful when joining between property tables and the tables denoting their domains and ranges. Note that these indexes and FKs not only optimise SQOWL2 in terms of query processing, but also type inference, in particular when triggers verify the queries specified in their ⟨condition⟩ fields.

Conventional Schema

As we have discussed in Section 4.6.1, when a given ontology contains FunctionalProperty, InverseFunctionalProperty and HasKey axioms, the canonical schema (i.e. the back-end schema $\mathsf{S_B}$) can be optimised to store information in what may be regarded as a more conventional schema. This allows SQOWL2 to be adapted so that it can be used to create triggers on top of existing relational schemas (as opposed to generating new schemas with associated triggers). In a nutshell, if we have i functional properties $P_1, \ldots, P_i$, all of which contain a class C as their PropertyDomain, j inverse functional properties $Q_1, \ldots, Q_j$ containing the same C as their PropertyRange, k key properties $R_1, \ldots, R_k$ which together uniquely identify individuals in the same C, then we store the class and properties as a single table:

$$C(\underline{id}, P_1, \ldots, P_i, Q_1, \ldots, Q_j, R_1, \ldots, R_k)$$

7.3 Evaluation of SQOWL2

This section evaluates SQOWL2 by comparing it to a non-materialising reasoner, Stardog and a materialising reasoner, OWLim, in order to experimentally verify the hypothesis that

materialising inference results leads to faster query processing at the expense of slower data updating. We chose Stardog and OWLim mainly because these two state-of-the-art reasoners provide community versions which are free for evaluation. Unlike SQOWL2, which uses an RDBMS as its data storage, the two comparison reasoners both store and access their data outside of an RDBMS (i.e. in a file system provided by the operating system), and consequently they do not perform inference in a transactional manner. Indeed, we believe SQOWL2 is unique in providing transactional reasoning combined with materialisation of the inferred data. In the evaluation, all experiments were processed on a machine with Intel i7-2600 CPU @ 3.40GHz, 8 Cores, and 16GB of memory, running Microsoft SQL Server 2014. SQOWL2 used OWL API v3.4.3 for ontology loading and Pellet v2.3.1 for classification. For the comparison reasoners, we used OWLim-Lite v5.4.6486 and Stardog-Community v2.2.1.

We evaluated the three systems by comparing their speed of data loading, data deleting and query processing. Each experiment was repeated 10 times, for which the average value is reported. The benchmark data we used is the well-known LUBM, which describes the knowledge in a university domain. LUBM was chosen because it has been widely used for evaluating large inference systems [ZML$^+$06, LMZ$^+$07, PZH08, WED$^+$08, GWW$^+$15, PURDG$^+$12, BCH$^+$14, GWW$^+$15]. LUBM has a small T-Box, and the capability of customising the size of the data generated. The T-Box of LUBM consists of 43 OWL classes, 32 OWL properties, and approximately 200 axioms. Besides the T-Box, the benchmark provides 14 queries which are numbered as Q1 – Q14 in this chapter. LUBM also offers a data generator, which is able to create A-Boxes of factual data of different sizes. In this chapter, a set of A-Boxes containing n universities is denoted by LUBM-n, in which each university has approximately 100,000 class and property facts.

The second evaluation task of SQOWL2 is on the inference soundness and completeness. We used SQOWL2 to process the 14 LUBM queries, and compared the results to those of Pellet, which is a tableaux-based reasoner known to provide sound and complete answers to the 14 queries. However, since the 14 LUBM queries are not exhaustive enough [SGH10], in order to further test the completeness level of our system, we additionally utilised SQOWL2 to answer more exhaustive test suites generated by SyGENiA, a system used for empirically benchmarking

the completeness of a reasoner. For a given query and a T-Box, SyGENiA generates a test suite containing a number of A-Box files that check all possible inference logic which generates results to this query w.r.t. the T-Box. Therefore, for a query and a T-Box, successfully passing the test suite means a reasoner is complete to answer this query over any arbitrary A-Box data of the tested ontology.

7.3.1 Performance of Incremental Type Inference

Performance of Data Loading

We firstly used the LUBM data generator to produce four datasets LUBM-25, LUBM-50, LUBM-100 and LUBM-200 with increasing sizes. Then, we used each system to load the four datasets and recorded the time they required, which is shown in Table 7.1[1]. Results show that for each system loading time increased linearly from LUBM-25 to LUBM-200, which implies that all of the three systems provided scalable data loading.

Table 7.1: Data loading time (s)

	LUBM-25	LUBM-50	LUBM-100	LUBM-200
SQOWL2	583	1,115	2,133	4,465
OWLim	79	160	336	743
Stardog	14	26	51	n/a

Table 7.2 further provides the calculated speed of loading LUBM-25 – LUBM-200 by the three inference systems, based on the results shown in Table 7.1.

Table 7.2: Data loading speed (inserts/s)

	LUBM-25	LUBM-50	LUBM-100	LUBM-200
SQOWL2	5,684	5,966	6,176	5,978
OWLim	42,067	41,623	39,946	35,933
Stardog	242,323	262,752	271,332	n/a

As can be seen, the speed of loading different sizes of data by each system was stable; for example, SQOWL2 was able to load data at approximately 6,000 inserts/s for all datasets.

[1]Since the size of LUBM-200 exceeds the license limitation of Stardog-Community, we do not include related performance results in Table 7.1 – Table 7.5 by writing n/a for not available.

The fastest one among the three systems was Stardog, because the system implements the query-rewriting approach, and there is no need to perform inference when loading the data. The slowest one was SQOWL2, because type inference by this system is performed as part of database transactions with full ACID properties (i.e. in a transactional manner), and the result of inference from loaded data is materialised in an RDBMS. We measured the overheads associated with providing ACID properties for database updates, and we found that even with no inference considered (i.e. we loaded the datasets with triggers disabled), the speed in the SQL Server database used for benchmarking was at about 14,600 inserts/s, which is slightly more than twice as faster as 6,000 inserts/s (i.e. the speed when SQOWL2 considered inference). Therefore, as compared to normal database operations in an RDBMS, the inference process provided by SQOWL2 causes a significant, but not impractical, overhead.

Performance of Data Deleting

After the four datasets (i.e. LUBM-25 – LUBM-200) have been loaded into the three inference systems, we used each of them to process several random deletes of A-Box data, and the performance is summarised in Table 7.3.

Table 7.3: Data deleting speed (deletes/s)

	LUBM-25	LUBM-50	LUBM-100	LUBM-200
SQOWL2	305	581	420	224
Stardog	29,268	28,759	28,903	n/a

In the above table, we only show the average speed of executing deletes by SQOWL2 and Stardog but omit the performance of OWLim, because the Lite version of OWLim does not handle data deletes in an incremental way, i.e. OWLim-Lite computes and materialises the whole inference again if any data is deleted. Indeed, as we recorded, even over the smallest LUBM-25, using OWLim to delete a random data item needed approximately 15 seconds for not only deleting this item of data, but also re-computing the inference closure. However, as shown in Table 7.3, SQOWL2 on average was able to process 305 deletes per second over LUBM-25, which was about 3ms for each deletion. Due to the fact that query-rewriting approaches do not perform inference when inserting or deleting facts, the speed of data deleting over the

four datasets by Stardog was stable, and also was much faster than a materialised approach SQOWL2, which incrementally updates the inference materialisation.

Moreover, it needs to be noticed that the label & check process decreased the speed of handling deletes by SQOWL2 when processing deletes from smaller datasets in comparison with larger ones. For example, SQOWL2 was able to perform about 581 deletes/s over LUBM-50 in comparison with 420 deletes/s over LUBM-100. However, there is an exception to this, which was from LUBM-25 to LUBM-50, SQOWL2's data deleting speed increased from 305 deletes/s to 581 deletes/s, and the reason as investigated was because the RDBMS switched to a more efficient execution plan. Because of the cost of type inference, SQOWL2 caused a significant overhead when comparing with itself without providing inference (but with indexes created), whose speed of processing deletes was at approximately 20,000 deletes/s.

7.3.2 Performance of Query Processing

For query processing, we converted each LUBM query from the format of **SPARQL** [HSP13] to an SQL query (or a Spark query programme shown in Section 7.5.2)[2]. Here we take Q1 as an example to show how its equivalent SQL query is executed by SQOWL2.

Figure 7.11 shows Q1 in SPARQL, which queries for some individuals `?X` which are members of the class GraduateStudent and are related by the property takesCourse to the individual `<http://www.Department0.University0.edu/GraduateCourse0>`.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type ub:GraduateStudent .
  ?X ub:takesCourse
<http://www.Department0.University0.edu/GraduateCourse0>}
```

Figure 7.11: LUBM Query 1 in SPARQL

The SQL query converted from this SPARQL query is given in Figure 7.12. Since both ex-

[2]A complete list of the 14 LUBM queries in SPARQL can be found in `http://swat.cse.lehigh.edu/projects/lubm/queries-sparql.txt`, and the translation of them to SQL and Spark can be found in `https://github.com/yl12510/thesis/tree/master/lubm`.

plicit and implicit facts of GraduateStudent and takesCourse are respectively materialised in tables GraduateStudent and takesCourse, to compute the answers, the SQL query essentially need to perform a join between the two tables on the columns `GraduateStudent.id` and `takesCourse.domain`, and the values of the column `takesCourse.range` should be equal to `'<http://www.Department0.University0.edu/GraduateCourse0>'`.

```
SELECT GraduateStudent.id
FROM   GraduateStudent
JOIN   takesCourse
ON     GraduateStudent.id = takesCourse.domain
AND    takesCourse.range =
       '<http://www.Department0.University0.edu/GraduateCourse0>'
```

Figure 7.12: LUBM Query 1 in SQL

Soundness & Completeness

In order to empirically evaluate the soundness and completeness of our system, instead of comparing to the two rule-based systems Stardog and OWLim, we compare SQOWL2 to a complete reasoner Pellet. We used SQOWL2 to answer the 14 LUBM queries over LUBM-1, and answers to the queries generated by SQOWL2 are exactly the same as the complete reasoner Pellet. Therefore, our system is sound and complete for processing the 14 queries over LUBM-1. However, this simple evaluation has two disadvantages. First, in order to compare with a tableaux-based reasoner, the datasets should be limited to a suitable size (e.g. LUBM-1) that can be handled by the tableaux-based reasoner. Secondly, the A-Box data generated by LUBM is argued to be not exhaustive enough in [GMSH12], because the data might not cover all inference logic that generates answers to a query. Thus, we cannot be sure whether our system is sound and complete when answering LUBM queries w.r.t. any arbitrary A-Boxes.

To further evaluate the soundness and completeness (especially the completeness, as soundness can be trivially verified by checking that each trigger produces valid outputs from valid inputs) of our system, we further used SQOWL2 to process the test suites generated by SyGENiA for the LUBM T-Box and the 14 queries. In a nutshell, for the T-Box and every query, SyGENiA generates a test suite containing a number of A-Boxes, each of which represents an inference case that generates answers to this query; for instance, the test suite for Q6 has 169 A-Boxes.

Therefore, the completeness level of an inference system for answering a query w.r.t. a T-Box can be calculated as dividing the number of passed inference cases in this test suite by the number of all inference cases contained in this test suite. In our experiments, SQOWL2 passed all inference cases of every test suite which SyGENiA generates for each LUBM query; therefore it is complete to answer the 14 queries w.r.t. the T-Box and any arbitrary LUBM A-Boxes. As found in [SGH10], the most complete rule-based system out of four evaluated was OWLim. Our tests show OWLim still only provides incomplete answers to Q6, Q8 and Q10 at the completeness level of 0.96, 0.93 and 0.96 respectively. Take Q6 as an example, the incompleteness is because OWLim partially supports the inference which includes existential quantification.

Efficiency

To evaluate the efficiency of query processing, we launched each of the three inference systems to simulate a load of cycling through the 14 LUBM queries over its persisted LUBM A-Box data, and afterwards recorded the time used for executing each query. The average speed of answering queries, i.e. how many queries can be processed in a minute by each of the three systems is shown in Table 7.4.

Table 7.4: Average query processing speed (q/m)

	LUBM-25	LUBM-50	LUBM-100	LUBM-200
SQOWL2	1,253	949	519	36
OWLim	446	229	102	42
Stardog	39	17	5	n/a

SQOWL2 or OWLim processed queries much faster than Stardog, since the two materialising-based systems compute and materialise explicit and implicit data before the stage of query processing. For example, the average speed of executing LUBM queries over LUBM-100 by SQOWL2 was approximately 100 times as fast as Stardog. If we only compare the two materialising systems, the average query answering speed by SQOWL2 was significantly faster than OWLim for datasets LUBM-25, LUBM-50 and LUBM-100, and was comparable to OWLim over the data of LUBM-200. SQOWL2 experienced a sharp drop from LUBM-100 to LUBM-

200 mostly because of Q2 (which required only 481ms over LUBM-100 but needed about 21s over LUBM-200). We checked the query plans used by the RDBMS for Q2 over LUBM-200 and found that it used the Nested Loops plan for table joins, which is less efficient than the Hash Match plan used for answering the same query over LUBM-100. Note that results shown in Table 7.4 is what we recorded without intentional tuning, but it is not abnormal in database applications that certain queries might need manual tuning by a database administrator, especially when they are executed over larger datasets.

Table 7.5: Detailed query processing time (ms)

LUBM-n	System	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14
LUBM-25	SQOWL2	0.0	248	0.0	2.0	7.2	31	1.0	70	277	0.0	3.4	12	1.0	17
	OWLim	168	202	1.2	5.4	7.8	97	2.2	404	922	1.2	1.4	28	2.2	39
	Stardog	57	259	52	85	57	19,172	279	681	243	394	56	77	56	53
LUBM-50	SQOWL2	0.0	196	0.0	2.2	7.0	25	0.8	116	489	0.0	7.0	4.0	2.0	36
	OWLim	278	512	1.6	5.6	7.8	164	2.0	698	1,867	1.0	1.4	43	1.8	80
	Stardog	56	492	53	91	59	46,084	506	1,213	445	701	55	85	56	52
LUBM-100	SQOWL2	0.0	481	0.0	2.0	7.2	38	1.0	112	931	0.0	14	4.2	5.4	22
	OWLim	476	1,969	1.0	5.4	8.0	290	2.6	1,368	3,831	1.2	1.8	86	2.4	155
	Stardog	57	830	55	90	58	157,022	879	2,178	831	1,106	57	89	57	52
LUBM-200	SQOWL2	0.0	21,040	0.0	2.2	8.0	57	1.2	133	1,855	0.0	2.0	5.2	12	44
	OWLim	900	7,872	1.6	6.2	8.2	472	2.2	2,623	7,639	1.6	1.2	178	4.2	315

We present more detailed results in Table 7.5[3], which lists the average execution time for every query by each of the three systems benchmarked. As can be seen, SQOWL2 was faster than OWLim in most cases over every LUBM dataset. Q11 is the only query which SQOWL2 always executed more slowly than OWLim, and its processing time by SQOWL2 increased from 56ms over LUBM-25 to 57ms over LUBM-100, but was suddenly decreased to just 2ms over LUBM-200 (which is nearly as fast as 1.2ms required by OWLim). The reason for this is that, the RDBMS used by SQOWL2 chose a more efficient Nested Loops plan for joining tables when processing Q11 over LUBM-200 rather than the less efficient Merge Join plan used over LUBM-25 to LUBM-100. Q2 is another interesting query we want to address. Apart from the significant drop from LUBM-100 to LUBM-200, the reason for which we have already explained in the last paragraph, SQOWL2 processed it faster than OWLim over both LUBM-50 and LUBM-100,

[3]In Table 7.5 and Table 7.6, SQOWL2's results are reported from Microsoft SQL Server. We round the average query processing times which are less than 10ms to one decimal place, and round the results which are greater than 10m to whole numbers.

but slightly slower over LUBM-25 and even significantly slower over LUBM-200. SQOWL2 processed this query over LUBM-50 even faster than over the smaller LUBM-25, because the RDBMS switched to a more efficient Parallelism plan for LUBM-50 (also for LUBM-100 and LUBM-200), but not over LUBM-25. Finally, Q13 is the last query which SQOWL2 did not dominate the performance over OWLim; indeed, SQOWL2 outperformed OWLim only over LUBM-25, but was sightly slower than OWLim over other datasets. The query plans used by the RDBMS for Q13 as we investigated did not change when the size of datasets grew; however, the processing time of this query by SQOWL2 did increase linearly.

When comparing SQOWL2 to Stardog, the former was much faster for all LUBM queries over LUBM-25, LUBM-50 and LUBM-100 except Q9, for which SQOWL2 processed slightly slower than Stardog. The reason for this is that the test suite generated by SyGENiA for Q9 only contains one inference case requiring joining just three tables; however, the original Q9 processed by SQOWL2 requires five joins among six tables. Stardog was significantly slower when answering Q6 and Q10 than both SQOWL2 and OWLim, because the two queries are rewritten to many sub-queries (the test suites for Q6 and Q10 have 169 and 168 inference cases, respectively), which are very complex to compute the answers.

Tuning SQOWL2 for Faster Query Processing

As we have mentioned in Section 7.2.3, we create for each class and property table indexes, which reduce the time used by the RDBMS to fetch required data in the table, and consequently the performance of query processing and incremental type inference can be accelerated. Although in our current implementation, we did not provide a systematic way of tuning the RDBMS which materialises the inference closure, we might manually apply certain optimisations; for example, the query processing performance can be significantly improved by adding SQL FKs or by forcing the RDBMS to choose a more efficient query plan.

Table 7.6 provides the improved query processing performance by SQOWL2 after we manually tuned the RDBMS (where '-' denotes that there was not a significant improvement). We may start from Q9 to demonstrate our tuning strategies. Q9 is the only query which SQOWL2

Table 7.6: Query improvements (ms) by SQOWL2 after database tuning (results in Table 7.5 → improved performance)

SQOWL2	Q2	Q5	Q7	Q9	Q13
LUBM-25	248 → 123	7.2 → 0.0	1.0 → 0.0	277 → 241	1.0 → 0.0
LUBM-50	-	7.0 → 0.0	0.8 → 0.4	489 → 431	2.0 → 0.0
LUBM-100	-	7.2 → 0.0	1.0 → 0.8	931 → 811	5.4 → 0.0
LUBM-200	21,040 → 1,723	8.0 → 0.0	1.2 → 1.0	1,855 → 1,640	12 → 0.0

processed slower than Stardog. We added SQL FKs (described below) which represent the semantics of PropertyDomain and PropertyRange of the property teacherOf.

$$S_B.\text{teacherOf}(\text{domain}) \overset{fk}{\Rightarrow} S_B.\text{Faculty}(\text{id})$$

$$S_B.\text{teacherOf}(\text{range}) \overset{fk}{\Rightarrow} S_B.\text{Course}(\text{id})$$

The two FKs optimised the query plan, especially for joining tables Faculty, Course and teacherOf used by the RDBMS. After adding the above two FKs, we then averaged the query processing time for Q2, which became 241ms over LUBM-25, 431ms over LUBM-50 and 811ms over LUBM-100. Comparing the new timings with Stardog illustrates that our system was now slightly faster than Stardog over all of the four LUBM datasets. Moreover, since the PropertyDomain of the property memberOf and the PropertyRange of the property hasAlumnus are both set as the class Person, we created two extra FKs shown as follows:

$$S_B.\text{memberOf}(\text{domain}) \overset{fk}{\Rightarrow} S_B.\text{Person}(\text{id})$$

$$S_B.\text{hasAlumnus}(\text{range}) \overset{fk}{\Rightarrow} S_B.\text{Person}(\text{id})$$

Adding these new FKs benefits Q5, Q7 and Q13 and as we tested processing them required much less time than without these FKs. As shown in Table 7.6, most of their processing times were reduced to less than 0.05ms, such as Q13 over LUBM-200, in which case SQOWL2 became faster than OWLim.

With regard to Q2, one FK displayed below capturing the semantics of the PropertyRange of undergraduateDegreeFrom can be added:

$$S_B.\text{undergraduateDegreeFrom}(\text{range}) \overset{fk}{\Rightarrow} S_B.\text{University}(\text{id})$$

This added FK caused the RDBMS to choose the more efficient Hash Match plan when executing Q2 over LUBM-25. However, for processing Q2 over LUBM-50 and LUBM-100, adding this

FK surprisingly decreased the processing time, as the RDBMS adapted itself 'wrongly' to use a less efficient plan, which is Nested Loops. It is not uncommon to witness 'wrong' self-education in an RDBMS, and after forcing the query to be executed by using instead the Hash Match plan, the quicker performance was re-obtained.

Moreover, we recorded the time required for processing this query over LUBM-200, when forcing the RDBMS to use the Hash Match plan, and the processing time was improved to approximately 1.7s. The new timing was much faster than the 21s used before, and became even much faster than 7.871s required by OWLim.

Besides PropertyDomain and PropertyRange axioms, other axioms constructed by using constructors SubClassOf and SubPropertyOf also result in new FKs. Note that these FKs are unnecessary otherwise, since our triggers for these axioms enforce the constraint implied from the FKs to be always satisfied. Therefore, at the stage of data uploading and deleting, we can temporarily disable the FK checks (or make them deferrable in Postgres), as constraint checks because of FKs usually slow down data loading and deleting.

7.4 Implementation of SPOWL

In Chapter 6, we have described how Spark programmes are individually compiled from each OWL 2 RL axiom by some transformation rules. These Spark programmes are executed as a Spark application on a cluster of machines. The Spark application computes and materialises the inference closure for a given ontology by executing its Spark programmes in an optimised order presented in Section 6.4.3, and should terminate if there is no new derivation.

Figure 7.13 outlines the components of a Spark application, which consists of a **driver** running on the master node of the cluster, and several **executors** running on worker nodes of the cluster. When a Spark application is executed, the driver first initialises a `SparkContext`, which defines RDDs and computation jobs (i.e. transformations and actions) specified on these RDDs by the Spark programmes. The `SparkContext` then distributes the computation jobs to executors, which process individual tasks that make up these jobs and return results back to

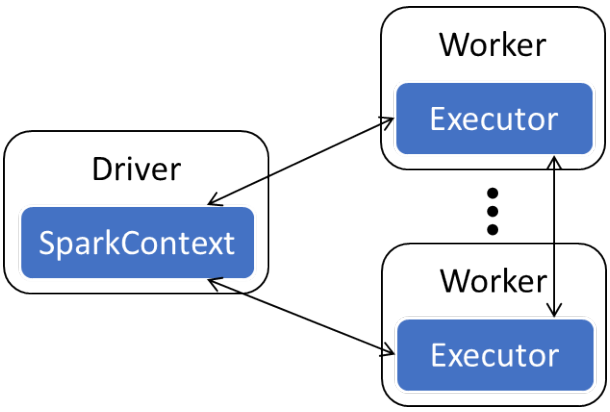

Figure 7.13: The Components of a Spark application

the `SparkContext`.

7.4.1 Spark Application Initialisation

In order to launch a Spark application, we first need a `SparkConf` to set up some configurations, such as the address of the master node where the driver will be running, and the application name. Then a `SparkContext` can be instantiated based on this `SparkConf`. A sample Spark code for initialising a Spark application is:

```
conf=SparkConf().setMaster($master).setAppName($appName)
sc=SparkContext(conf=conf)
```

Besides setting the `$master` and `$appName`, Spark also allows the specifications of other configurations, such as the amount of memory allocated to the driver and executors, and the amount of parallelism for executing this application.

7.4.2 Initial Data Loading

With the `SparkContext`, we are now ready to load the explicit A-Box of facts into a distributed file system, which is an HDFS in SPOWL. Assume `$localABox` denotes the path where the original A-Box files are stored in a local file system, we can use the `textFile` function in Spark to load them as an `ABox_RDD`:

```
ABox_RDD = sc.textFile("file://$localABox")
```

Next, we may call another Spark function `saveAsTextFile` to simply store `ABox_RDD` in the HDFS (assume `$hdfsABox` is the HDFS path where we expect to store):

```
ABox_RDD.saveAsTextFile("hdfs://$hdfsABox")
```

such that A-Box facts are loaded from the local file system to the HDFS. However, saving `ABox_RDD`, which contains all class and property facts, as a union in the HDFS leads to an inefficient data access, because each time we access instances of a particular class or property, we need to filter out a fragment from the `ABox_RDD`. Therefore, we instead perform the filtering process only once, and store instances of different classes and properties separately in the HDFS, so that no further filter process of class and property instances is required. This benefits not only computing the type inference closure, but also computing answers to queries over the ontological data.

To specify the filter function, it is worth mentioning that ontologies often present their A-Box facts as **Resource Description Framework** (**RDF**) triples [CWL14] of the format (s, p, o). Triples which express class facts (called **class triples**) are in the format of $\langle a \text{ rdf:type } C \rangle$, where rdf:type is a keyword for specifying that an individual a is a member of a class C, while property facts are specified by **property triples** as $\langle a\ P\ b \rangle$, which denotes a is related to b by a property P. Thus $\langle a \text{ rdf:type } C \rangle$ and $\langle a\ P\ b \rangle$ are the RDF way of asserting A-Box facts $C(a)$ and $P(a, b)$ in DL, respectively.

Thus to filter out instances of a class C as `$C_RDD`, we select from `$ABox_RDD` the triples (s, p, o) where o is equal to C and p is equal to rdf:type, and for filtering out instances of a property P as `$P_RDD`, we select the triples (s, p, o) where p is equal to P:

```
$C_RDD = ABox_RDD.filter(lambda (s,p,o):o == $C).map(lambda (s,p,o):s)
$P_RDD = ABox_RDD.filter(lambda (s,p,o):p == $P).map(lambda (s,p,o):(s,o))
```

Note that, since we are only interested in s (i.e. the individuals) from class triples, and (s, o) (i.e. the pairs of property subject and object) from the property triples, two `map` transformations for selecting these fields are performed in the above code.

Afterwards, `$C_RDD` and `$P_RDD` can be stored in the HDFS by:

```
$C_RDD.saveAsTextFile("hdfs://$hdfsABox/$C/EXP")
$P_RDD.saveAsTextFile("hdfs://$hdfsABox/$P/EXP")
```

Note that, as `$C_RDD` or `$P_RDD` now only contain the explicit facts, we store them in paths with suffix "`EXP`" to differentiate the paths suffixed with "`IMP`" used for storing implicit inference.

7.4.3 Physical Spark Programmes Generation

Spark programmes have already been described in a logical manner in Chapter 6, and we now show how they are implemented into real code. Because we have already used the Python syntax for specifying logical Spark programmes, the implementation of them to physical code is trivial. Take the transformation rule for a SubClassOf axiom $C \sqsubseteq D$ as an example, the logical Spark programmes $D_{rdd} = D_{rdd}.\mathsf{union}(C_{rdd})$ is implemented into the physical programmes as:

```
$C_RDD = sc.textFile("hdfs://$hdfsABox/$C")
$D_RDD = sc.textFile("hdfs://$hdfsABox/$D")
$D_IMP = $C_RDD.subtract($D_RDD)
$D_RDD = $D_RDD.union($D_IMP)
```

where the first two lines respectively read instances of C and D as `$C_RDD` and `$D_RDD` (no filtering process is required here). As paths are not suffixed with the "`EXP`" or "`IMP`", `$C_RDD` and `$D_RDD` will include both explicit and implicit items of data. The third line computes inference, which should be added to `$D_RDD` (i.e. `$C_RDD.subtract($D_RDD)` which includes data in `$C_RDD` but not in `$D_RDD`). The final line then merges `$D_RDD` with `$D_IMP`.

In essence, we respectively create `$C_IMP` and `$P_IMP` for each class C and property P. They can be used for checking whether new inference is derived for every C or P (by checking whether `$C_IMP` or `$P_IMP` is empty or not), which is essential for SPOWL to decide when to terminate the whole Spark application. The Spark application should stop if for all classes C and properties P, their `$C_IMP` and `$P_IMP` are empty. For a non-empty `$C_IMP` or `$P_IMP`, we store them containing inference results in the path suffixed with "`IMP`" by (i.e. inference is materialised):

```
$C_IMP.saveAsTextFile("hdfs://$hdfsABox/$C/IMP")
$P_IMP.saveAsTextFile("hdfs://$hdfsABox/$P/IMP")
```

In addition to showing the physical Spark programmes for $C \sqsubseteq D$, we further list the code for an AllValuesFrom axiom $C \sqsubseteq \forall P.D$:

```
$P_RDD = sc.textFile("hdfs://$hdfsABox/$P")
$C_RDD = sc.textFile("hdfs://$hdfsABox/$C")
$D_RDD = sc.textFile("hdfs://$hdfsABox/$D")
$D_IMP = $C_RDD.map(lambda xc: (xc,xc)).join($P_RDD).map(lambda (xk,(xc,yp)):yp)
$D_IMP = $D_IMP.subtract($D_RDD)
$D_RDD = $D_RDD.union($D_IMP)
```

which performs a join between `$C_RDD` with `$P_RDD`, in order to compute data items that should be added to `$D_RDD`. Finally, we list the code which implements the logical Spark programmes for a TransitiveProperty P (i.e. $P \circ P \sqsubseteq P$):

```
$P_RDD = sc.textFile("hdfs://$hdfsABox/$P/")
TC = $P_RDD
# start the loop for computing transitive closure
while True:
  $P_TMP = TC.map(lambda (xp,yp):(yp,xp)).join(TC).map(lambda (yk,(xp,zp)):(xp,zp))
  if $P_TMP.isEmpty():
     break
  TC = TC.union($P_TMP)
#loop finished
$P_IMP = TC.subtract($P_RDD)
$P_RDD = TC
```

The above physical Spark programmes initialise an RDD `TC` as equal to `$P_RDD`, and after each iteration of computing new transitive pairs, `TC` is merged with the new computed pairs. Thus when the loop of transitive closure computation is finished, `TC`, which contains the transitive closure of P, is assigned to `$P_RDD`.

7.4.4 Optimisation in SPOWL

- **Caching Data in Memory**

  In the physical Spark programmes, we cache an RDD which is repeatedly used to avoid re-computation of this RDD. For instance, for an IntersectionOf axiom $C \sqsubseteq C_1 \sqcap \ldots \sqcap C_n$, because a tableaux-based reasoner classifies it to n SubClassOf axioms (i.e. $C \sqsubseteq C_1, \ldots, C \sqsubseteq C_n$), the data of `$C_RDD` (representing C) will be used n times; therefore,

caching `$C_RDD` by a Spark function called `cache` or `persist` in memory will improve the performance of reasoning this axiom.

- **Partitioning before Join**

  In Spark, a normal join between two sets of key-value pairs will shuffle the pairs whose keys are the same to the same executor, so that joint pairs can be computed. However, in the case of one set of key-value pairs is very large while the other set is quite small, this normal join might result in a slow shuffle process because of shuffling the large set. Instead, we partition the large set of key-value pairs by their keys, and only copy the small set to the node where each partition of the large set is stored. Since this reduces the amount of data transferred through the network of a cluster, the join can be performed much faster. Partitioning an RDD can be achieved by a Spark function `partitionBy`.

- **Using Hadoop Sequence Files**

  As we use an HDFS for a distributed data storage, we can store the inference materialisation as Hadoop sequence files rather than text files. Thus the inference materialisation can be compressed and be approached as binary files, which results in more efficient storage and computation. Spark provides a function called `saveAsSequenceFile` for storing an RDD as Hadoop sequence files, which can be read by a `SparkContext` through another function named `sequenceFile`.

7.5 Evaluation of SPOWL

In this section, we evaluate the scalability of SPOWL for inference materialisation and query processing. All experiments were performed on a cluster of 9 machines running on a private cloud environment[4] containing a master node (with CPU @ 2.5GHz, 4 Cores, and 16GB of Memory), and 8 slave nodes (each with CPU @ 2.5GHz, 4 Cores, and 16 GB of Memory). The cluster ran Hadoop version 2.6.0-cdh5.5.0 (with 2.08 TB configured capacity), and Apache Spark 1.6.0. SPOWL used OWL API v3.4.3 for T-Box loading, and supports the use of Pellet

[4] `https://www.doc.ic.ac.uk/csg/services/cloud`

v2.3.1 or Hermit v1.3.8 for T-Box classification. We do not provide a comparison between SPOWL and other large reasoning systems, due to difficulty of configuring and running them on the private cloud.

As for the evaluation of SQOWL2, we use the LUBM data generator, but now to create much larger datasets. Specifically, we produced five new datasets, which are LUBM-400, LUBM-800, LUBM-1200, LUBM-1600 and LUBM-2000, where the largest one (i.e. LUBM-2000) contains approximately 270 million A-Box facts and is about 44GB in size. Similarly to the evaluation of SQOWL2, each experiment was repeated 10 times, of which the average value is reported.

7.5.1 Performance of Inference Materialisation

- **Initial Load**: We used SPOWL to first load the original A-Boxes for each LUBM dataset, during which stage instances of every class or property were filtered out and materialised in separate folders in the HDFS. We recorded the time which SPOWL used for loading each dataset in Table 7.7.

Table 7.7: Performance of Inference Materialisation by SPOWL

SPOWL	LUBM-400	LUBM-800	LUBM-1200	LUBM-1600	LUBM-2000
Initial Load	9m08s	20m30s	27m50s	41m20s	54m10s
Type Inference	10m19s	16m28s	33m20s	38m58s	58m08s
Total Time	19m27s	36m58s	1h01m10s	1h20m18s	1h52m18s

  As can be seen, the time used by SPOWL increased almost linearly when loading datasets from LUBM-400 to LUBM-2000. In particular, SPOWL was able to initially load LUBM-2000 (having about 270 million facts of 44GB) in 55 minutes (i.e. the loading speed was at about 81,818 facts/s). We may highlight this linear increase by translating the results in Table 7.7 into a line chart in Figure 7.14.

- **Type Inference**: After the step of initial loading, we used SPOWL to perform type inference by launching a Spark application which executed Spark programmes generated from T-Box axioms over the loaded data. The results of type inference were computed and materialised in the HDFS. The time used by SPOWL for the five datasets is shown in

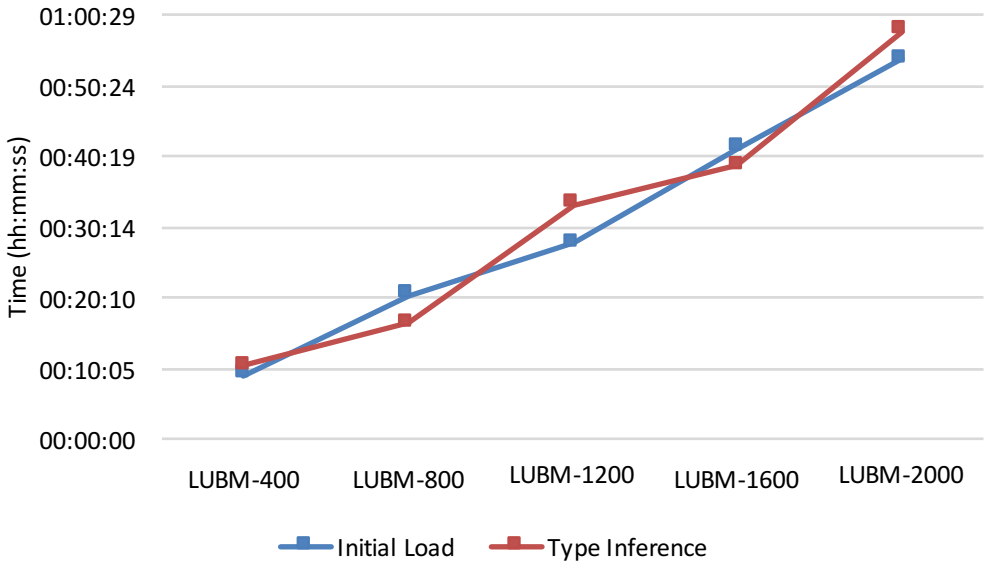

Figure 7.14: Inference Materialisation by SPOWL

Table 7.7, and is also available in the line chart shown in Figure 7.14. Again, the time used for type inference grew almost linearly from LUBM-400 to LUBM-2000. Furthermore, for the largest dataset LUBM-2000, SPOWL materialised about 246 million implicit facts in 59 minutes (i.e. at the speed of 70,690 facts/s).

7.5.2 Performance of Query Processing

In order to process the 14 LUBM queries by SPOWL, we translated the SPARQL LUBM query into a Spark query programme. We take Q1 as an example, and the Spark query programme for this query is shown in Figure 7.15.

```
# read data from the two csv files
GraduateStudent = sc.textFile("hdfs://$hdfsABox/GraduateStudent")
takesCourse = sc.textFile("hdfs://$hdfsABox/takesCourse")
ran = "<http://www.Department0.University0.edu/GraduateCourse0>"
# select pairs (s,o) from takesCourse where o is equal to ran
domTakesCourse = takesCourse.filter(lambda (s,o) : o == ran).map(lambda (s,o) : s)
# compute the final answer to this query
ans = domTakesCourse.intersection(GraduateStudent)
```

Figure 7.15: LUBM Query 1 in Spark

As can be seen, in the Spark programme implementing LUBM Q1, both explicit and implicit instances of GraduateStudent and takesCourse are read from the HDFS as RDDs `GraduateStudent`

and `takesCourse`. Remember that because we store inference results separately for each class and property, no filtering process of each class or property from the whole inference is required. Then, we need to first select from `takesCourse` the pairs whose objects are equal to `<http://www.Department0.University0.edu/GraduateCourse0>`. Finally, the intersection of `GraduateStudent` and the subjects from the selected pairs are computed as the answers.

Table 7.8: Performance of Query Processing by SPOWL

LUBM	Avg.	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14
400	23s	47s	1m07s	1m05s	12s	16s	13s	10s	5s	12s	2s	3s	27s	10s	32s
800	40s	1m12s	2m14s	1m56s	20s	34s	21s	21s	5s	18s	3s	5s	47s	10s	1m01s
1200	1m12s	1m45s	4m36s	4m12s	42s	58s	32s	35s	5s	28s	4s	7s	1m12s	10s	1m20s
1600	2m07s	2m25s	7m12s	5m56s	2m21s	1m34s	1m20s	1m52s	5s	40s	9s	21s	2m44s	13s	2m50s
2000	3m30s	4m26s	10m46s	8m53s	3m38s	2m52s	2m32s	3m34s	6s	1m18s	26s	28s	4m44s	12s	5m12s

In order to test the performance of query processing, we cycled through the 14 queries (in Spark) over the inference materialisations of LUBM-400 – LUBM-2000 by using SPOWL, and the performance results are provided in Table 7.8. As can be seen, besides detailed processing time for each query over each dataset, the average query processing time (i.e. how much time on average SPOWL required to process each query) are shown in the column Avg., which is translated to a line chart in Figure 7.16.

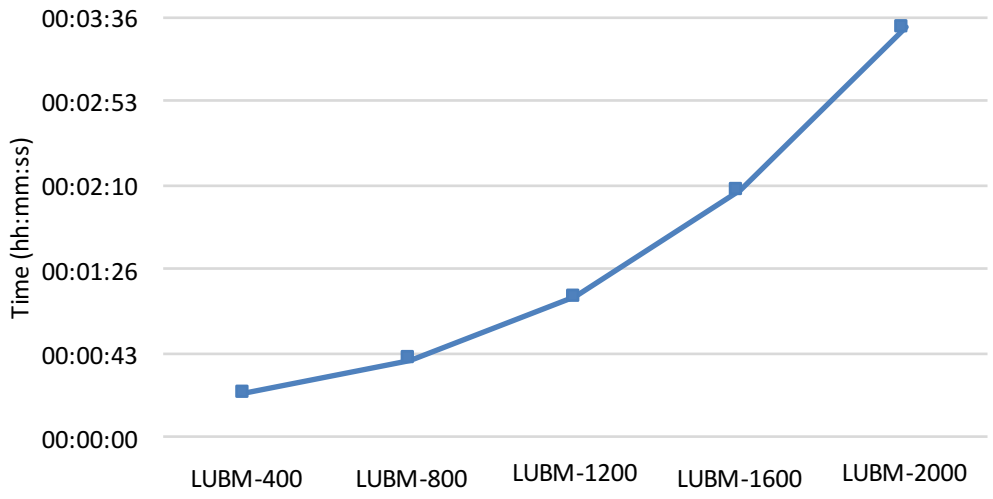

Figure 7.16: Average Query Processing by SPOWL

Obviously the average query processing time was increased beyond linearly. However, to further determine which type of growth the average performance follows, we respectively calculated the

$n \log n$ and n^2 values w.r.t. the size of LUBM datasets, and then divided each set of the $n \log n$ and n^2 values by the average query processing time. We then analysed which ratio computed is more constant, and found the average query processing follows a quadratic growth (i.e. $\mathcal{O}(n^2)$); in other words, the performance of average query processing is proportional to the square of the size of the input LUBM dataset.

Although this is not a linear increase, the average query performance by SPOWL does not violate the PTIME-complete complexity stated by OWL 2 standard for conjunctive query processing in OWL 2 RL. The average performance is better than PTIME-complete due to the fact that the LUBM T-Box is relatively simple although it contains some additional axioms that surpass the OWL 2 RL profile. Moreover, some of the 14 LUBM queries are quite trivial to process, such as Q8 and Q13. This also averages the overall query performance of SPOWL. Indeed, by further applying this analysis method to the time used for processing each query, we found for most of the queries, their processing time follows a quadratic growth, except Q8 and Q13 which follow a logarithmic growth (i.e. $\mathcal{O}(\log n)$).

7.6 Summary

In this chapter, we have described the implementation of SQOWL2 which performs type inference in an RDBMS by using triggers, and SPOWL, which conducts type inference in a Big Data system by Spark programmes. In particular, we have shown how logical triggers and logical Spark programmes can be implemented into real code, alongside some optimisations which can further improve the performance of the two implementations.

Furthermore, we have presented the evaluation of the two implementations. For SQOWL2, we used it to process LUBM datasets that are small enough to store in an RDBMS, and evaluated the performance of handling data inserts and deletes, and the completeness and efficiency of query processing. We have verified that SQOWL2 was able to provide a faster query processing and a more complete inference completeness than the two comparison systems, Stardog and OWLim, at the expense of slower data updates mostly because of providing transactional

reasoning. For SPOWL, we mainly focus on its scalability of materialising type inference results for large ontologies. In our evaluation, for initial data loading and type inference, SPOWL scaled linearly to process LUBM-2000 which contains 270 million explicit A-Box facts; and for query processing, the average performance by SPOWL followed a quadratic growth. It should be noted that the cluster used for evaluating SPOWL contains only 9 nodes, and we believe SPOWL can handle even larger ontologies if more nodes are added.

Some limitations of the evaluation should be addressed. Firstly, the soundness and completeness of SQOWL2 was only experimentally tested for the T-Box and A-Boxes provided by LUBM. Indeed, other benchmarks, such as **University Ontology Benchmark** (**UOBM**) [MYQ+06] and **Berlin SPARQL Benchmark** (**BSBM**) [BS09], can be used for further evaluation. Secondly, when comparing SQOWL2 to Stardog and OWLim, the performance differences might result from the data access and data storage mechanisms used by them. For data access, SQOWL2 uses an RDBMS, which normally performs certain optimisations for query processing, as compared to a file system used by Stardog and OWLim. However, an RDBMS also conducts constraint checking, which consequently slows the data updating of SQOWL2.

Chapter 8

Conclusion

In this thesis, we have presented our approach to managing inference over large OWL ontological data in mainstream databases. This approach is feasible to be used in existing database applications to enhance them with inference capabilities. We particulally focus on applications which use a small schema to represent their large scale data, and are more query-intensive than update-intensive. By materialising the inference results, our approach provides faster query processing than non-materialising approaches, at the expense of slower data updating. Depending on the size of inference materialisation, our work materialises the result of inference for OWL 2 ontologies with a simple T-Box but large A-Boxes in an RDBMS or in a Big Data system. By separating the T-Box and A-Box, our work is able to apply a tableaux-based reasoner for a complete T-Box classification w.r.t. the T-Box. Axioms in the classified T-Box are transformed into an inference framework containing active and non-active rules for type inference over large A-Boxes. This combination of tableaux-based and rule-based inference computes and stores for each instance its membership of classes and properties before any query is executed, so that querying the ontological data requires no real-time inference. This work is conjectured to be a sound and complete implementation of the OWL 2 RL/RDF rules for type inference over OWL 2 RL ontologies, as rules in the inference framework cover not only all OWL 2 RL/RDF rules (except rules which conflict the UNA followed by our work), but also some extra OWL 2 axioms.

Our work can be viewed as a compiler which implements the inference framework in different

programming languages. When the ontology is small enough to be processed in an RDBMS, our work compiles the active ECA rules into triggers, and we term this part of work SQOWL2. In SQOWL2, triggers invoked by updates to the database will analyse how the previous type inference results should be updated accordingly. This process is also known as transactional reasoning, which preserves the important ACID properties of database transactions. SQOWL2 performs type inference from both data inserts and deletes in an incremental manner without needing to re-compute the type inference results. Especially for handling deletes, triggers invoked by them automatically conduct a label & check process, which first labels all items of data affected by the deletes, and then checks as to whether the labelled data can be re-inferred from non-labelled data or not. Among state-of-the-art reasoners (e.g. Stardog, DLDB, Ontop, OWLim, RDFox, WebPIE, Oracle's RDF Store and Minerva), we believe SQOWL2 is unique in providing transactional reasoning combined with incrementally maintaining the materialisation of the inferred data from both data inserts and deletes.

From the implementation viewpoint, SQOWL2 gathers together all logical trigger fragments associated with each table and implements them as a single physical trigger, to improve the performance. Moreover, SQOWL2 adopts some optimisations, such as adding indexes and FKs. It is also possible for an RDBMS administrator to perform standard tuning of the RDBMS to use a more efficient execution plan, which accelerates not only inference computation but also query processing.

In the case of ontologies that are too large to be processed in an RDBMS, our work compiles the if-then part of the trigger rules into Spark programmes in a Big Data system, and we name it SPOWL. It executes Spark programmes which are directly translated from a classified T-Box. This avoids unnecessary and inefficient rule-matching, often witnessed by most large-scale reasoners (e.g. WebPIE and Cichlid), which simply evaluate a fixed set of entailment rules for computing inference. Spark programmes are non-active, and should be executed iteratively by a Spark application, in order to compute and materialise the type inference results until no implicit data can be inferred. To minimise the number of executing iterations, an optimised order based the bottom-up hierarchy of the ontology T-Box is followed.

In the implementation of SPOWL, since it adopts Spark as the computation framework, it caches datasets which are frequently used in memory as much as possible, in order to avoid the needing to recompute these datasets. Moreover, when performing a join between a large dataset and a small one, we partition the large set and shuffle only the small set to each partition (i.e. shuffling the large set is avoided), in order to obtain a more efficient performance.

We have also evaluated SQOWL2 and SPOWL over the well-known benchmark LUBM. For evaluating SQOWL2, we have empirically tested its soundness and completeness of query processing, results of which show it was complete for answering all LUBM queries w.r.t. LUBM T-Box and any arbitrary LUBM A-Boxes, which outperformed most rule-based reasoners [SGH10]. We have further compared the performance of processing data inserts, data deletes and queries by SQOWL2 to two comparison reasoners, Stardog (a query-rewriting approach) and OWLim (a materialised approach). Results show SQOWL2 was faster at query answering, at the expense of slower data updating speed mostly because of providing transactional reasoning.

In the evaluation of SPOWL, we have used it to materialise the inference closure for much larger LUBM datasets, and to process LUBM queries over the materialisation. Materialising type inference and processing queries were both scaled up to run over LUBM-2000, which has about 270 million A-Box facts. Since the cluster used in the evaluation only contains 9 nodes, we believe SPOWL can scale up to reasoning over even larger ontologies by adding more nodes to the cluster.

8.1 Summary of Thesis Achievements

Our work extends the previous work SQOWL, and the extensions provide the following genernal contributions to knowledge about inference:

1. SQOWL2 improves SQOWL, which only performs type inference from data inserts in an RDBMS, to now support type inference from also data deletes. In SQOWL2, we improve the existing DRed algorithm to assign for each item of ontological data a state, which represents its semantics for being persisted in the RDBMS. When handling inserts,

SQOWL2, when compared to DRed, not only computes type inference but also properly updates the data state, which is essential for our new label & check process to handle deletes. Updates are captured by triggers, which perform type inference in a transactional and incremental manner. We have published this part of work in [LM15, LM16].

2. SQOWL2 further extends SQOWL to support type inference for OWL 2 ontologies as compared to the previous support of OWL 1. This thesis thus contributes additional triggers to cover constructs in OWL 2 not found in SQOWL's triggers for OWL 1. It should be noted that our work not only covers the OWL 2 RL/RDF rules (except those rules that disobey the UNA) but also some extra OWL 2 semantics. We conjecture SQOWL2 is a sound and complete implementation of the OWL 2 RL/RDF rules for type inference over OWL 2 RL ontologies. This part of work has been published in [LM13, LM14].

3. SPOWL further extends SQOWL2 to support type inference over ontologies with A-Boxes that are too large to be processed by an RDBMS. In SPOWL we contribute a new technique, where each OWL 2 axiom in an ontology is compiled into a corresponding Spark programme. These Spark programmes are executed iteratively in a bottom-up order following the T-Box hierarchy. Compared to existing approaches, this has the advantage that the order of execution is optimised for each ontology, instead of following a fixed execution order. SPOWL addresses scalable type inference over large OWL 2 RL ontologies, and so it adopts a Big Data system rather than an RDBMS. A consequence of this is that SPOWL does not provide the transactional reasoning supported by SQOWL2.

8.2 Future Work

Of course, there are some areas in which our work can be improved in future efforts, especially with respect to the limitations which have been discussed though the thesis.

First, our work takes the assumption that the ontology T-Box is small enough for a tableaux-based reasoner to classify, and thus benefit from their complete T-Box inference. However,

with respect to large T-Boxes (e.g. Gene ontology and SNOMED-CT), our approach should be extended to use a non-tableaux reasoner that aims for large T-Box inference.

Second, our work stores a total materialisation of the type inference results, in order to provide faster query-processing than non-materialising approaches. However, the total materialisation can sometimes take too much space for data storage. From this viewpoint, our work can be extended to only materialise the inference results that are frequently queried, by combining query-rewriting to process queries over less frequently accessed facts.

Third, our work of SQOWL2 only supports updates to A-Boxes, but not to the T-Box, by assuming the T-Box is static. Thus, we can extend our work to support updates to the whole ontology. This could be achieved by applying a similar label & check process which we have used for data updates to incrementally handle T-Box updates.

References

[ABB+00] Michael Ashburner, Catherine A Ball, Judith A Blake, David Botstein, Heather Butler, J Michael Cherry, Allan P Davis, Kara Dolinski, Selina S Dwight, Janan T Eppig, et al. Gene Ontology: Tool for the Unification of Biology. *Nature genetics*, 25(1):25–29, 2000.

[Abb12] Sunitha Abburu. A Survey on Ontology Reasoners and Comparison. *International Journal of Computer Applications*, 57(17), 2012.

[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 *The semantic web*, pages 722–735. Springer-Verlag, Berlin Heidelberg, 2007.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[Bar77] Jon Barwise. An Introduction to First-Order Logic. *Studies in Logic and the Foundations of Mathematics*, 90:5–46, 1977.

[Bat89] Carlo Batini. Entity-Relationship Approach: A bridge to the User. In *7th International Conference on on Enity-Relationship Approach*, 1989.

[BB93] Alex Borgida and Ronald J Brachman. Loading Data into Description Reasoners. In *ACM SIGMOD Record*, volume 22, pages 217–226. ACM, 1993.

[BCH+14] Timea Bagosi, Diego Calvanese, Josef Hardi, Sarah Komla-Ebri, Davide Lanti, Martin Rezk, Mariano Rodríguez-Muro, Mindaugas Slusnys, and Guohui Xiao. The Ontop Framework for Ontology Based Data Access. In *Proceedings of CSWS*

2014, pages 67–77, Wuhan, China, 8–12 August 2014. Springer-Verlag, Berlin Heidelberg.

[BCM+10] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications (Second Edition)*. Cambridge University Press, Cambridge, UK, 2010.

[BCN92] Carlo Batini, Stefano Ceri, and Shamkant B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings, 1992.

[Boo05] Grady Booch. *The Unified Modeling Language User Guide*. Pearson Education India, 2005.

[BS09] Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

[CDGL+07] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. *Journal of Automated reasoning*, 39(3):385–429, 2007.

[Cod70] Edgar F Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Con15] The UniProt Consortium. UniProt: a Hub for Protein Information. *Nucleic Acids Research*, 43(D1):D204–D212, 2015.

[CWL14] Richard Cyganiak, David Wood, and Markus Lanthaler, editors. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation, 2014. Latest version available at `https://www.w3.org/TR/rdf11-concepts/`.

[Dat00] Chris J. Date. *An Introduction to Database Systems (7. ed.)*. Addison-Wesley-Longman, Boston, 2000.

[DB87] M Darnovsky and J Bowman. Transact-SQL User's Guide. Technical report, 1987.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DK09] Vincent Delaitre and Yevgeny Kazakov. Classifying ELH Ontologies In SQL Databases. In *Proceedings of OWLED 2009*, Chantilly, VA, 23–24 October 2009. CEUR-WS.org.

[DLSW99] Guozhu Dong, Leonid Libkin, Jianwen Su, and Limsoon Wong. Maintaining Transitive Closure of Graphs in SQL. *International Journal of Information Technology*, 51(1):46, 1999.

[Don06] Kevin Donnelly. SNOMED-CT: The advanced terminology and coding system for eHealth. *Studies in health technology and informatics*, 121:279, 2006.

[Doy79] Jon Doyle. A Truth Maintenance System. *Artif. Intell.*, 12(3):231–272, 1979.

[Gär03] Peter Gärdenfors. *Belief Revision*, volume 29. Cambridge University Press, 2003.

[Geo11] Lars George. *HBase: the Definitive Guide*. " O'Reilly Media, Inc.", 2011.

[GHM+08] Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter Patel-Schneider, and Ulrike Sattler. OWL 2: The Next Step for OWL. *J. Web Sem.*, 6(4):309–322, 2008.

[GHM+14] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. HermiT: An OWL 2 Reasoner. *J. Autom. Reasoning*, 53(3):245–269, 2014.

[GHP+12] Bernardo Cuenca Grau, Ian Horrocks, Bijan Parsia, Alan Ruttenberg, and Michael Schneider. *OWL 2 Web Ontology Language: Mapping to RDF Graphs*. W3C Recommendation, 2012. Latest version available at `https://www.w3.org/TR/owl2-mapping-to-rdf/`.

[GHVD03] Benjamin N Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proceedings of WWW 2003*, pages 48–57, Budapest, Hungary, 20–24 May 2003. ACM Press, New York, USA.

[GM05] Stephan Grimm and Boris Motik. Closed World Reasoning in the Semantic Web through Epistemic Operators. In *OWLED*, 2005.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. Maintaining Views Incrementally. In *Proceedings of ACM SIGMOD 93*, pages 157–166, Washington, D.C., USA, 26–28 May 1993. ACM Press, New York, USA.

[GMSH12] Bernardo Cuenca Grau, Boris Motik, Giorgos Stoilos, and Ian Horrocks. Completeness Guarantees for Incomplete Ontology Reasoners: Theory and Practice. *J. Artif. Intell. Res. (JAIR)*, 43:419–476, 2012.

[Goo16] Google. Freebase Data Dumps. `https://developers.google.com/freebase`, 2016.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2):158–182, 2005.

[GWW+15] Rong Gu, Shanyong Wang, Fangfang Wang, Chunfeng Yuan, and Yihua Huang. Cichlid: Efficient Large Scale RDFS/OWL Reasoning with Spark. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 700–709. IEEE, 2015.

[HKR09] Pascal Hitzler, Markus Krotzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. CRC Press, 2009.

[Hor97] Ian R Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, Citeseer, 1997.

[HPS14] Patrick Hayes and Peter Patel-Schneider. *RDF 1.1 Semantics*. W3C Recommendation, 2014. Latest version available at `http://www.w3.org/TR/rdf11-mt/`.

[HR83] Theo Härder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[HSP13] Steve Harris, Andy Seaborne, and Eric Prudhommeaux, editors. *SPARQL 1.1 query language*. W3C Recommendation, 2013. Latest version available at `https://www.w3.org/TR/sparql11-query/`.

[KGZ99] Kevin Kline, Lee Gould, and Andrew Zanevsky. *Transact-SQL Programming: Covers Microsoft SQL Server 6.5/7.0 and Sybase Adaptive Server 11.5*. O'Reilly Media, Sebastopol, CA, 1999.

[KKWZ15] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-Fast Big Data Analysis*. " O'Reilly Media, Inc.", 2015.

[KM91] Hirofumi Katsuno and Alberto O. Mendelzon. On the Difference between Updating a Knowledge Base and Revising It. pages 387–394, 1991.

[KMR10] Markus Krötzsch, Anees Mehdi, and Sebastian Rudolph. Orel: Database-Driven Reasoning for OWL 2 Profiles. In *Proceedings of DL 2010*, page 114, Waterloo, Ontario, Canada, 4–7 May 2010. CEUR-WS.org.

[KOM05] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. OWLIM–A Pragmatic Semantic Repository for OWL. In *Proceedings of WISE Workshops 2005*, pages 182–192, New York, NY, 20–22 November 2005. Springer-Verlag, Berlin Heidelberg.

[Krö12a] Markus Krötzsch. OWL 2 Profiles: An Introduction to Lightweight Ontology Languages. In *Proceedings of Reasoning Web - Semantic Technologies for Advanced Query Answering - 8th International Summer School 2012*, Vienna, Austria, 3–8 September 2012. Springer-Verlag, Berlin Heidelberg.

[Krö12b] Markus Krötzsch. The Not-So-Easy Task of Computing Class Subsumptions in OWL RL. In *Proceedings of ISWC 2012*, pages 279–294, Boston, MA, 11–15 November 2012. Springer-Verlag, Berlin Heidelberg.

[KSH12] Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. A Description Logic Primer. *CoRR*, 2012.

[KWE10] Vladimir Kolovski, Zhe Wu, and George Eadon. Optimizing Enterprise-Scale OWL 2 RL Reasoning in a Relational Database System. In *International Semantic Web Conference*, pages 436–452. Springer-Verlag, Berlin Heidelberg, 2010.

[Leh92] Fritz Lehmann. Semantic Networks. *Computers & Mathematics with Applications*, 23(2-5):1–50, 1992.

[LM11] A Lakshman and P Malik. The Apache Cassandra Project. *Retrieved from the World Wide Web October*, 31:2014, 2011.

[LM13] Yu Liu and Peter McBrien. SQOWL2: Transactional Type Inference for OWL 2 DL in an RDBMS. In *Informal Proceedings of DL 2013*, pages 779–790, Ulm, Germany, 23–26 July 2013. CEUR-WS.org.

[LM14] Yu Liu and Peter McBrien. SQOWL2: Ontology-based Data Access for OWL 2 RL in an RDBMS. Technical report, Automed, 2014.

[LM15] Yu Liu and Peter McBrien. Transactional and Incremental Type Inference from Data Updates. In *Proceedings of BICOD 2015*, pages 206–219, Edinburgh, UK, 6–8 July 2015. Springer–Verlag, Berlin Heidelberg.

[LM16] Yu Liu and Peter McBrien. Transactional and Incremental Type Inference from Data Updates. *The Computer Journal*, 2016.

[LMZ+07] Jing Lu, Li Ma, Lei Zhang, Jean-Sébastien Brunner, Chen Wang, Yue Pan, and Yong Yu. SOR: a practical system for ontology storage, reasoning and search. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1402–1405. VLDB Endowment, 2007.

[LRU14] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014.

[LTW09] Carsten Lutz, David Toman, and Frank Wolter. Conjunctive Query Answering in the Description Logic EL Using a Relational Database System. In *Proceedings of IJCAI 2009*, volume 9, pages 2070–2075, Pasadena, California, 11–17 July 2009. AAAI Press, Palo Alto, California.

[MB08] Georgios Meditskos and Nick Bassiliades. Combining a DL Reasoner and a Rule Engine for Improving Entailment-Based OWL Reasoning. In *Proceedings of ISWC 2008*, pages 277–292, Karlsruhe, Germany, 26–30 October 2008. Springer-Verlag, Berlin Heidelberg.

[MB10] Georgios Meditskos and Nick Bassiliades. DLEJena: A practical forward-chaining OWL 2 RL reasoner combining Jena and Pellet. *J. Web Sem.*, 8(1):89–94, 2010.

[MGH+12] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz, editors. *OWL 2 Web Ontology Language: Profiles (Second Edition)*. W3C Recommendation, 2012. Latest version available at `http://www.w3.org/TR/owl2-profiles/`.

[Mil95] George A Miller. WordNet: a Lexical Database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[Min82] Jack Minker. On Indefinite Databases and the Closed World Assumption. In *International Conference on Automated Deduction*, pages 292–308. Springer-Verlag, Berlin Heidelberg, 1982.

[MNP+14] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In *Proceedings of AAAI 2014*, pages 129–137, Québec City, Québec, Canada, 27–31 July 2014. AAAI Press, Palo Alto, California.

[MNPH15] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Incremental Update of Datalog Materialisation: the Backward/Forward Algorithm. In *Proceedings of*

AAAI 2015, pages 1560–1568, Austin, Texas, USA, 25–30 January 2015. AAAI Press, Palo Alto, California.

[Mom01] Bruce Momjian. *PostgreSQL: Introduction and Concepts*, volume 192. Addison-Wesley, New York, 2001.

[Mot07] Boris Motik. On the Properties of Metamodeling in OWL. *J. Log. Comput.*, 17(4):617–637, 2007.

[MPSG12] Boris Motik, Peter F Patel-Schneider, and Bernardo Cuenca Grau, editors. *OWL 2 Web Ontology Language: Direct Semantics (Second Edition)*. W3C Recommendation, 2012. Latest version available at `http://www.w3.org/TR/owl2-direct-semantics/`.

[MRS09] P McBrien, N Rizopoulos, and A Smith. SQOWL: Performing OWL-DL type inference in SQL. Technical report, Citeseer, 2009.

[MRS10] PJ McBrien, Nikos Rizopoulos, and Andrew Charles Smith. SQOWL: Type Inference in an RDBMS. In *Proceedings of ER 2010*, pages 362–376, Vancouver, BC, Canada, 1–4 November 2010. Springer-Verlag, Berlin Heidelberg.

[MRS12] Peter McBrien, Nikos Rizopoulos, and Andrew Charles Smith. Type Inference Methods and Performance for Data in an RDBMS. In *Proceedings of SWIM 2012*, page 6, Scottsdale, AZ, USA, 20 May 2012. ACM Press, New York, USA.

[MS06] Boris Motik and Ulrike Sattler. A Comparison of Reasoning Techniques for Querying Large Description Logic ABoxes. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 227–241. Springer-Verlag, Berlin Heidelberg, 2006.

[MVH04] Deborah L McGuinness and Frank Van Harmelen, editors. *OWL Web Ontology Language Overview*. W3C Recommendation, 2004. Latest version available at `https://www.w3.org/TR/owl-features/`.

[MVPG09] Sergio Muñoz-Venegas, Jorge Pérez, and Claudio Gutierrez. Simple and Efficient Minimal RDFS. *J. Web Sem.*, 7(3):220–234, 2009.

[MYQ+06] Li Ma, Yang Yang, Zhaoming Qiu, Guo Tong Xie, Yue Pan, and Shengping Liu. Towards a Complete OWL Ontology Benchmark. In *ESWC*, volume 4011 of *Lecture Notes in Computer Science*, pages 125–139. Springer-Verlag, Berlin Heidelberg, 2006.

[NPM+15] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. RDFox: A Highly-Scalable RDF Store. In *International Semantic Web Conference (2)*, volume 9367 of *Lecture Notes in Computer Science*, pages 3–20. Springer-Verlag, Berlin Heidelberg, 2015.

[NY83] Jean-Marie Nicolas and Kioumars Yazdanian. An Outline of BDGEN: A Deductive DBMS. In *Proceedings of IFIP Congress*, pages 711–717, Paris, France, 19–23 September 1983. Springer–Verlag, Berlin Heidelberg.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110. ACM, 2008.

[PDR05] Chaoyi Pang, Guozhu Dong, and Kotagiri Ramamohanarao. Incremental Maintenance of Shortest Distance and Transitive Closure in First-Order Logic and SQL. *ACM Transactions on Database Systems (TODS)*, 30(3):698–721, 2005.

[PK03] Jan Paralic and Ivan Kostial. Ontology-based Information Retrieval. In *Proceedings of the 14th International Conference on Information and Intelligent systems (IIS 2003), Varazdin, Croatia*, pages 23–28, 2003.

[PUHM09] Héctor Pérez-Urbina, Ian Horrocks, and Boris Motik. Efficient Query Answering for OWL 2. In *International Semantic Web Conference*, pages 489–504. Springer-Verlag, Berlin Heidelberg, 2009.

[PURDG+12] Héctor Pérez-Urbina, Edgar Rodríguez-Díaz, Michael Grove, George Konstantinidis, and Evren Sirin. Evaluation of Query Rewriting Approaches for OWL

2. In *Proceedings of SSWS+HPCSW*, volume 943, pages 32–44, Boston, USA, 11 November 2012. CEUR-WS.org.

[PZH08] Zhengxiang Pan, Xingjian Zhang, and Jeff Heflin. DLDB2: A Scalable Multi-perspective Semantic Web Repository. In *Proceedings of WI-IAT'08*, volume 1, pages 489–495, Sydney, NSW, Australia, 9–12 December 2008. IEEE, New York.

[RASG07] Yves Raimond, Samer A Abdallah, Mark B Sandler, and Frederick Giasson. The Music Ontology. In *ISMIR*, volume 422. Citeseer, 2007.

[Rei78] Raymond Reiter. On Closed World Data Bases. In *Logic and data bases*, pages 55–76. Springer-Verlag, Berlin Heidelberg, 1978.

[Rud11] Sebastian Rudolph. Foundations of Description Logics. In *Reasoning Web. Semantic Technologies for the Web of Data*, pages 76–136. Springer-Verlag, Berlin Heidelberg, 2011.

[Sch12] Michael Schneider, editor. *OWL 2 Web Ontology Language: RDF-Based Semantics (Second Edition)*. W3C Recommendation, 2012. Latest version available at `http://www.w3.org/TR/owl2-rdf-based-semantics/`.

[SGH10] Giorgos Stoilos, Bernardo Cuenca Grau, and Ian Horrocks. How Incomplete is Your Semantic Web Reasoner? In *Proceedings of AAAI 2010*, Atlanta, Georgia, USA, 11–15 July 2010. AAAI Press, Palo Alto, California.

[Sow00] John F Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks Cole Publishing Co., Pacific Grove, CA, 2000.

[Sow14] John F Sowa. *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, 2014.

[SPG+07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007.

[SS11] Christian Seitz and René Schönfelder. Rule-based OWL Reasoning for Specific Embedded Devices. In *International Semantic Web Conference*, pages 237–252. Springer-Verlag, Berlin Heidelberg, 2011.

[tH05] Herman J ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *J. Web Sem.*, 3(2):79–115, 2005.

[TH06] Dmitry Tsarkov and Ian Horrocks. FaCT++ Description Logic Reasoner: System Description. In *Proceedings of IJCAR 2006*, pages 292–297, Seattle, WA, 17–20 August 2006. Springer-Verlag, Berlin Heidelberg.

[TPR10] Edward Thomas, Jeff Z Pan, and Yuan Ren. TrOWL: Tractable OWL 2 Reasoning Infrastructure. In *Extended Semantic Web Conference*, pages 431–435. Springer-Verlag, Berlin Heidelberg, 2010.

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[UKM+12] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank Van Harmelen, and Henri Bal. WebPIE: A Web-scale Parallel Inference Engine using MapReduce. *J. Web Sem.*, 10:59–75, 2012.

[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 *Proceedings of ISWC 2013*, pages 657–672, Sydney, NSW, Australia, 21–25 October 2013. Springer-Verlag, Berlin Heidelberg.

[UVHSB11] Jacopo Urbani, Frank Van Harmelen, Stefan Schlobach, and Henri Bal. QueryPIE: Backward Reasoning for OWL Horst over Very Large Knowledge Bases. In *International Semantic Web Conference*, pages 730–745. Springer-Verlag, Berlin Heidelberg, 2011.

[VSM03] Raphael Volz, Steffen Staab, and Boris Motik. Incremental Maintenance of Materialized Ontologies. In *CoopIS/DOA/ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 707–724. Springer-Verlag, Berlin Heidelberg, 2003.

[WED+08] Zhe Wu, George Eadon, Souripriya Das, Eugene Inseok Chong, Vladimir Kolovski, Melliyal Annamalai, and Jagannathan Srinivasan. Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle. In *Proceedings of ICDE 2008*, pages 1239–1248, Cancún, México, 7–12 April 2008. IEEE, New York.

[Whi15] Tom White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (4. ed., revised & updated)*. O'Reilly Media, Sebastopol, CA, 2015.

[ZML+06] Jian Zhou, Li Ma, Qiaoling Liu, Lei Zhang, Yong Yu, and Yue Pan. Minerva: A Scalable OWL Ontology Storage and Inference System. In *Proceedings of ASWC 2006*, pages 429–443, Beijing, China, 3–7 September 2006. Springer-Verlag, Berlin Heidelberg.

Appendix A

List of Abbreviations

A-Box	**Assertional Box**
ACID	**Atomicity, Consistency, Isolation, Durability**
ATIDB	**Auto Type Inference Database**
BSBM	**Berlin SPARQL Benchmark**
DAG	**Directed Acyclic Graph**
DL	**Description Logic**
DLP	**Description Logic Program**
DRed	**Delete & Rederive**
ECA	**Event Condition Action**
ER	**Entity-Relationship**
FK	**Foreign Key**
FOL	**First-Order Logic**
HDFS	**Hadoop Distributed File System**
KR	**Knowledge Representation**
LUBM	**Lehigh University Benchmark**
NNF	**Negation Normal Form**
OBDA	**Ontology-based Data Access**

OWL	**Web Ontology Language**
PK	**Primary Key**
RDBMS	**Relational Database Management System**
RDD	**Resilient Distributed Dataset**
RDF	**Resource Description Framework**
T-Box	**Terminological Box**
UML	**Unified Modelling Language**
UNA	**Unique Name Assumption**
UOBM	**University Ontology Benchmark**
W3C	**World Wide Web Consortium**
3NF	**Third Normal Form**

Appendix B

Theorem PR1

Let $\mathcal{R}$ be the OWL 2 RL/RDF rules, and let $\mathcal{O}_1$ and $\mathcal{O}_2$ be OWL 2 RL ontologies satisfying the following properties:

- no IRI in $\mathcal{O}_1$ or $\mathcal{O}_2$ is used both as a class and an individual;
- $\mathcal{O}_1$ does not contain axioms of SubAnnotationPropertyOf, AnnotationPropertyDomain and AnnotationPropertyRange;
- each axiom in $\mathcal{O}_2$ is an assertion of the form $C(a)$, $P(a, b)$ or $=(a_1, \ldots, a_n)$.

Furthermore, let $\mathsf{RDF}(\mathcal{O}_1)$ and $\mathsf{RDF}(\mathcal{O}_2)$ be translations of $\mathcal{O}_1$ and $\mathcal{O}_2$, respectively, into RDF graphs as specified in [GHP+12]; and let $\mathsf{FO}(\mathsf{RDF}(\mathcal{O}_1))$ and $\mathsf{FO}(\mathsf{RDF}(\mathcal{O}_2))$ be the translation of these graphs into first-order theories. Then, $\mathcal{O}_1$ entails $\mathcal{O}_2$ under the Direct Semantics if and only if $\mathsf{FO}(\mathsf{RDF}(\mathcal{O}_1)) \cup \mathcal{R}$ entails $\mathsf{FO}(\mathsf{RDF}(\mathcal{O}_2))$ under the standard first-order semantics.