Answering Object Queries over Knowledge Bases with Expressive Underlying Description Logics

by

Jiewen Wu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2013

Abstract

Many information sources can be viewed as collections of objects and descriptions about objects. The relationship between objects is often characterized by a set of constraints that semantically encode background knowledge of some domain. The most straightforward and fundamental way to access information in these repositories is to search for objects that satisfy certain selection criteria. This work considers a description logics (DL) based representation of such information sources and object queries, which allows for automated reasoning over the constraints accompanying objects. Formally, a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ captures constraints in the terminology (a TBox) $\mathcal{T}$, and objects with their descriptions in the assertions (an ABox) $\mathcal{A}$, using some DL dialect $\mathfrak{L}$. In such a setting, object descriptions are $\mathfrak{L}$-concepts and object identifiers correspond to individual names occurring in $\mathcal{K}$. Correspondingly, object queries are the well known problem of *instance retrieval* in the underlying DL knowledge base $\mathcal{K}$, which returns the identifiers of qualifying objects.

This work generalizes instance retrieval over knowledge bases to provide users with answers in which both identifiers and *descriptions* of qualifying objects are given. The proposed query paradigm, called *assertion retrieval*, is favoured over instance retrieval since it provides more informative answers to users. A more compelling reason is related to performance: assertion retrieval enables a transfer of basic relational database techniques, such as caching and query rewriting, in the context of an assertion retrieval algebra.

The main contributions of this work are two-fold: one concerns optimizing the fundamental reasoning task that underlies assertion retrieval, namely, *instance checking*, and the other establishes a query compilation framework based on the assertion retrieval algebra. The former is necessary because an assertion retrieval query can entail a large volume of instance checking requests in the form of $\mathcal{K} \models a : C$, where a is an individual name and C is a $\mathfrak{L}$-concept. This work thus proposes a novel absorption technique, *ABox absorption*, to improve instance checking. ABox absorption handles knowledge bases that have an expressive underlying dialect $\mathfrak{L}$, for instance, that requires disjunctive knowledge. It works particularly well when knowledge bases contain a large number of concrete domain concepts for object descriptions.

This work further presents a query compilation framework based on the assertion retrieval algebra to make assertion retrieval more practical. In the framework, a suite of rewriting rules is provided to generate a variety of query plans, with a focus on plans that avoid reasoning w.r.t. the background knowledge bases when sufficient cached results of earlier requests exist. ABox absorption and the query compilation framework have been implemented in a prototypical system, dubbed CARE Assertion Retrieval Engine (CARE).

CARE also defines a simple yet effective cost model to search for the best plan generated by query rewriting. Empirical studies of CARE have shown that the proposed techniques in this work make assertion retrieval a practical application over a variety of domains.

Acknowledgements

I would not have been able to complete my Ph.D. dissertation without the support of many people over the past five years. First, I owe my deepest gratitude and respect to my supervisors, Professor David Toman and Professor Grant Weddell. Their scholarship, attention to detail, enthusiasm, and immense knowledge have helped me in all stages of my doctoral studies.

My thanks are also due to my committee members for their numerous insightful suggestions for the final revision of this dissertation: Professors Peter van Beek, Leopoldo Bertossi, Lukasz Golab, and Richard Trefler, and to my collaborators for their contributions: Dr. Jeffrey Pound, Taras Kinash, and Dr. Alexander K. Hudek.

I am indebted to my colleagues in the database research group and the faculty and staff members in the School of Computer Science for their outstanding teaching, wise advices, interesting seminars, and superb services, which have made my Ph.D. journey entertaining and relaxing. I also want to thank all my friends for their support throughout the five years.

Most importantly, I would like to thank my parents, Fengchun Wu and Liuying Tian, and my brother Jixiong Wu, for their constant love, understanding, and care.

Table of Contents

List of Algorithms **xiii**

List of Abbreviations **xv**

1 Introduction **1**

1.1 Contributions 7

1.2 Organization 9

2 Background and Related Work **11**

2.1 The Resource Description Framework 12

2.1.1 Extensions to RDF 14

2.2 Description Logics 18

2.3 Query Answering over Knowledge Bases 27

2.3.1 The SPARQL Query Language 27

2.3.2 Conjunctive Queries 30

2.3.3 Ontology Based Data Access 31

2.4 Query Optimization 36

2.4.1 Query Optimization in DL Knowledge Bases 39

3 User Queries and Query Plans **41**

3.1 Assertion Queries 41

3.2 The Projection Description 42
3.3 Comparing Query Languages 47
3.4 Computing Projection 49
3.4.1 Procedures ProjConcepts and ProjFeature 50
3.4.2 Procedure ProjJoin 52
3.4.3 Procedure ProjRole 52
3.4.4 Procedure Reduce 54
3.4.5 Correctness 54
3.5 A Query Plan Language 56
3.5.1 Description Index 57
3.5.2 The Algebra 60

4 ABox Absorption 65

4.1 Obtaining an ABox Absorption 67
4.1.1 Computing $\mathcal{T}^0_{\mathcal{K}}$ 68
4.1.2 Optimizing Local Universal Restrictions 71
4.1.3 Instance Checking as Subsumption Checking 75
4.2 On Generalized Binary Absorption 80
4.2.1 On Witnesses 80
4.2.2 On Binary Absorption 81
4.3 A Procedure for ABox Absorption 83
4.3.1 The Procedure 84
4.3.2 Correctness 86
4.4 An Empirical Evaluation 89
4.4.1 Datasets and Queries 90
4.4.2 Comparing Guarding Strategies 94
4.4.3 Comparing CARE with DL Reasoners 95
4.4.4 Optimizing Typing Constraints 104

5 Query Compilation **107**

5.1 Query Rewriting 109

5.1.1 Queries as Algebraic Expressions 109

5.2 $\mathcal{K}$-free Rewriting 111

5.2.1 Determining $\mathcal{K}$-free Rewritings 114

5.3 Query Generation and Selection in CARE 118

5.3.1 Implementing Operators in CARE 119

5.3.2 Plan Selection in CARE 128

5.4 An Empirical Evaluation 134

6 Conclusions and Future Research **139**

6.1 Future Research 140

References **143**

APPENDICES **153**

A Correctness Proofs of the Rewriting Rules **155**

List of Algorithms

1 PROJ($Pd, \mathcal{K}, Rp, a : C$) . . . 50

2 PROJCONCEPT($C_1, \mathcal{K}, Rp, a : C$) . . . 51

3 PROJFEATURE($f, \mathcal{K}, Rp, a : C$) . . . 51

4 ITERATIVESEARCH($f, \mathcal{K}, Rp, a : C, low, high, constants$) . . . 51

5 PROJJOIN($Pd_1, Pd_2, \mathcal{K}, Rp, a : C$) . . . 52

6 PROJROLE($Pd, R, \mathcal{K}, Rp, a : C$) . . . 53

7 REDUCE($S, \mathcal{K}, Rp$) . . . 54

8 `Open`($T, root, a : C$) (Search-based) . . . 121

9 `GetNext`() (Search-based) . . . 122

10 CHOOSENEXT($triple, itr$) . . . 123

11 `Open`($T, root, a : C$) (Traversal-based) . . . 124

12 `GetNext`() (Traversal-based) . . . 124

13 `Open`(S_i, `Q`) (Cache Scans) . . . 125

14 `GetNext`() (Cache Scans) . . . 125

15 `Open`($\mathtt{Q}_1, \mathtt{Q}_2$) (Join) . . . 126

16 `GetNext`() (Nested loops) . . . 127

17 `GetNext`() (Sort-merge) . . . 127

List of Abbreviations

ABox	Assertion Box
BGP	Basic Graph Pattern
CQ	Conjunctive Query
CWA	Close World Assumption
DL	Description Logic
FG	Full Guarding
IRI	Internationalized Resource Identifier
KB	Knowledge Base
KR	Knowledge Represnetation
NG	No Guarding
OBDA	Ontology-based Data Access
OWA	Open World Assumption
OWL	Web Ontology Language
PG	Partial Guarding
RDF	Resource Description Framework
RDFS	RDF Schema
TBox	Terminology Box
UNA	Unique Name Assumption

Chapter 1

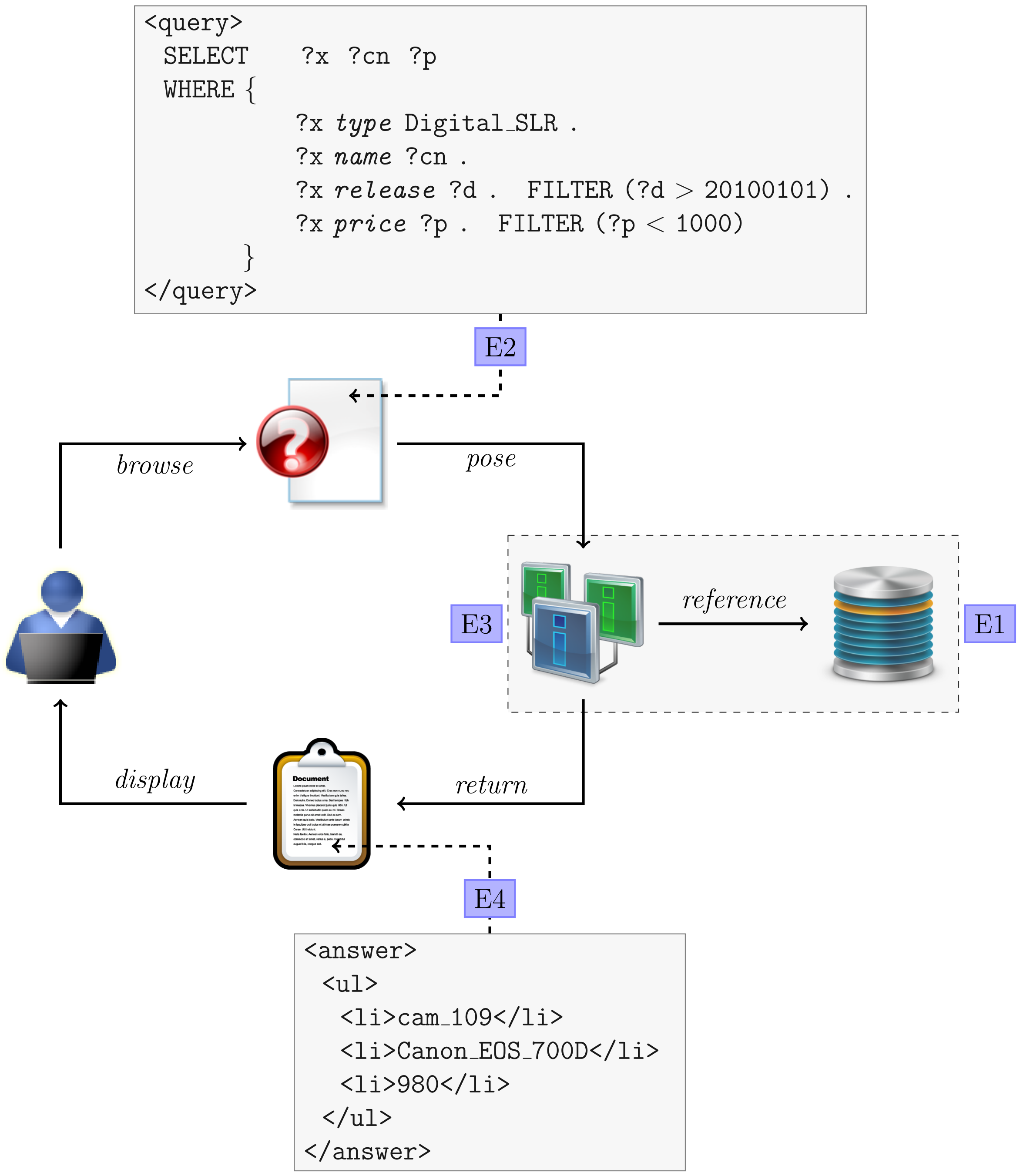
Introduction

But every error is due to extraneous factors (such as emotion and education); reason itself does not err.
— Kurt Gödel, November 29, 1972

A long-standing view of the structure of data in information management systems has been the relational schema; however, the Web provides a universal medium for exchanging information structured in diverse ways. Relational databases emphasize efficient access to structured data via some query language, supported by various query optimization techniques. The sharing of information on the Web relies on the textual structure of information, thus, adopting the information retrieval paradigm. In both scenarios, the two aspects of utmost importance are the structure used to represent information and the query language adopted to access information.

In the Web setting, information access is more challenging, largely due to the diversity of information structures and the absence of powerful query languages, as opposed to the mature relational technology. However, querying information remains the focal interest of users. Figure 1.1 illustrates a typical scenario in which Web users query specific information over some information repository. The following discussions revolve around this scenario and show how this dissertation work fits this general framework and why it is different from existing systems, such as relational databases.

In the scenario depicted in Figure 1.1, assuming an enterprise with some information repository (E1), a user browses some web page of this enterprise, for example, an HTML or XML page displaying company products, and may issue some requests during browsing, such as searching for specific products, which is transmitted to the enterprise web server(s)



```
<query>
 SELECT    ?x ?cn ?p
 WHERE {
            ?x type Digital_SLR .
            ?x name ?cn .
            ?x release ?d .  FILTER (?d > 20100101) .
            ?x price ?p .  FILTER (?p < 1000)
       }
</query>
```

```
<answer>
 <ul>
  <li>cam_109</li>
  <li>Canon_EOS_700D</li>
  <li>980</li>
 </ul>
</answer>
```

Figure 1.1: An application scenario.

in the form of a *query* (E2) that has been embedded in this page. The query, once sent to the server, is evaluated over the enterprise information repository (E1) by a query engine (E3). The computed answers to this query (E4) are ultimately rendered properly on some web page and presented to users.

Consider the case of an online site specialized in photography equipment. Following Figure 1.1, a web user browses an HTML/XML web page (E1) that lists all the camera products available for purchase. In particular, the user initiates a search event, which activates an embedded query on the web page that encodes the following information:

```
Return the camera names and prices of digital SLR cameras that are
      released after January 1, 2010 and sold for less than $1000.
```
(1.1)

In this situation, fulfilling the user's request requires the application system to address several important issues, such as how a user query is posed and how query answers are computed. These problems are characterized by understanding the four highlighted artefacts, E1-E4, in Figure 1.1, which are elaborated subsequently.

Information Repository (E1)

A typical realization of E1 is a relational database, in which an enterprise stores all its information in well-structured relations. For the Web, *linked data* [Heath and Bizer, 2011] has emerged as the mainstream approach to sharing information, particularly for the Semantic Web. The rational behind linked data is to use a standard mechanism for referring to things and connecting them. The mechanism for this purpose has been established by the Resource Description Framework (RDF), which is one of the NoSQL endeavours to the new generation of non-relational databases. More importantly, RDF is so far the only NoSQL solution that embraces the linked data technology. As can been seen in Section 2.1, RDF allows data to be represented as *triples*, each having a subject, predicate and object. Such a representation is simple and straightforward but is well suited to the Web architecture.

RDF data models also address a fundamental aspect of data: the semantics. In relational databases, the so-called *closed world assumption* (CWA) is adopted. Under CWA, data is considered to be complete and definite. For instance, the enterprise in Figure 1.1, were it to use relational databases, would consider any camera products not explicitly in its databases to be *non existent*. In lieu of CWA, RDF models use the *open world assumption* (OWA) for *incomplete* information. In this case, the enterprise information repository admits missing camera products and acknowledges the existences of such products. This subject will be revisited with a detailed discussion in Chapter 2.

RDF models can be enriched by more expressive languages (see Section 2.1.1) to represent domain knowledge, prominently the Web Ontology Language (OWL) [Hitzler et al., 2012]. An information repository represented by OWL is often called an OWL ontology, which uses a commonly-shared vocabulary with well-grounded semantics to model a domain in an *object-oriented* manner. In Figure 1.1, E1 can be an ontology maintained by the enterprise, which captures background knowledge of cameras and facts about particular camera objects available for purchase through various partner resellers. Similar to a relational database, the knowledge in an ontology can be either *intensional* (the schema) or *extensional* (the data); for instance, the ontology can state that

$$\texttt{all mirrorless cameras are digital SLR cameras} \tag{1.2}$$

as part of its schema and that

$$\texttt{the camera product } cam_{101} \texttt{ is a mirrorless camera} \tag{1.3}$$

as part of its data. In this dissertation, the term *ontology* is used interchangeably with the term *knowledge base*.

This work considers application systems that operate on information repositories represented as OWL ontologies. Note that ontologies are formalized by the OWL language. The full OWL is so expressive that inferences over OWL may be undecidable. In practice, however, we believe many application systems can choose a fairly expressive sublanguage of the full OWL for data modelling. Particularly, a class of languages that has the ability to express *indefinite* knowledge, i.e., non-deterministic domain knowledge, is investigated in this work.

Query (E2)

E2 in Figure 1.1 denotes the vehicle that an application uses to access information stored in the repository: queries. There exist a variety of query languages, depending on the information sources to be queried over. In relational systems, the de-facto query language is SQL, which has its root in first-order logic (FOL). For RDF repositories, a similar language to SQL has been widely used: the SPARQL query language (see Section 2.3.1). E2 in Figure 1.1 is a SPARQL query that formulates the user query expressed by the English sentence in (1.1). The query itself is easy to understand, possibly with the exception of the predicate ***type***, which, in this case, denotes *class membership*, i.e., any object that must be a `Digital_SLR` camera. In this application, the only complication is that this SPARQL query is embedded in a web page. Yet this is no departure from standard web applications that use relational systems, since there could be an embedded SQL query that is sent to the web server by the browser.

Since E1 is assumed to be an ontology, of which the modelling subscribes to an object-oriented view of some application domain, it is natural to view E1 as an information repository about objects. In a typical enterprise information system, employees, events, and projects, among others, are often viewed as objects. In Figure 1.1, for instance, the online site maintains camera products as objects. Thus, the query (1.1) looks for some objects (products) that satisfy the given conditions. Such queries are considered to be *object queries* in that their purpose is to retrieve objects that satisfy certain conditions.

Object queries can have comparatively simple forms, which can be easily understood and formulated by users, thus, many online sites support object queries as the main mechanism for querying. Since the query (1.1) requires the query answers to include camera names and prices, there is a need for a projection-style operation. This work then considers, roughly, a subset of the SPARQL query language to generalize object queries. As introduced in Chapter 3, this subset of SPARQL queries, called *assertion queries*, can be used to retrieve not only objects but *descriptions about objects*. The semantics of assertion queries differs from that of the corresponding SPARQL queries in that assertion queries can compute genuine nested relational values and handle the so-called *non-distinguished* variable (or existential variables). We will revisit the differences between assertion queries and SPARQL queries in Section 3.3, when sufficient background is introduced.

Query Processor (E3)

Query optimization, though transparent to end users, is essential for ensuring query performance. In relational DBMSs, query processing usually consists of multiple phases to translate a SQL query into a query plan that takes advantage of the underlying physical design for efficient query evaluation, as depicted in Figure 1.2. A SQL query in relational

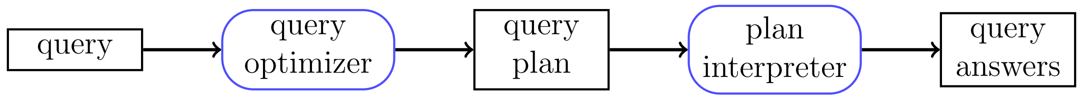


Figure 1.2: An overview of query evaluation in relational databases.

DBMSs is often translated into at least one query plan, because there is always a primary or secondary index associated with every relation referenced in the query. For SPARQL queries, the performance issues have been dealt with mainly over RDF datasets, using relational-style techniques, for example, DB2 [Bornea et al., 2013], Oracle [Kolovski et al., 2010], and other research prototypes [Neumann and Weikum, 2010]. For ontologies that exploit more expressive languages, query optimization has only been cursorily studied. This dissertation work focuses on query answering over ontologies, in which inferences are necessary. One of the objectives of this work is to establish an optimization framework that takes advantage of existing relational query processing techniques, which can be applied to answering assertion queries over ontologies with expressive underlying description logics.

Query answering over knowledge bases has some peculiar features, compared to query evaluation in relational DBMSs. In databases, a schema is used to ensure the consistency of data, or to obtain query plans, but is usually not considered during query execution (i.e., the plan interpretation phase in Figure 1.2). In contrast, a schema will be exploited during the whole process of query evaluation over knowledge bases. Recall that the ontology E1 permits missing or indefinite information, the query processor has to reason about schema bearing ontological knowledge to correctly answer a query. Given the query (1.1) and the statement (1.2) in the schema of E1, when evaluating the query over E1, a query processor must include all mirrorless cameras as part of the answers despite the query only asking for digital SLR cameras.

Inferences over an OWL ontology with expressive underlying logics are computationally intensive, hence, query answering over a knowledge base is more challenging than that over relational databases. A query processor for answering object queries over knowledge bases must have an ability to minimize or even avoid reasoning with ontological knowledge. In addition, when inferences are inevitable, it must be able to improve query answering. This can be achieved by devising new optimization techniques and adapting relational query optimization techniques, as further discussed in Chapters 4 and 5.

Query Answer (E4)

For a SQL query, the returned answers are in the from of relations. SPARQL queries return answers in RDF triples, which can also be thought of as relations. E4 in Figure 1.1 depicts a possible query answer to the query in E2. Recall that the query (1.1) requests the query answers be a combination of the identifiers and descriptions about objects. The query answer is described by a HTML/XML code snippet, which consists of the object identifier of a returned object and a list of descriptions of this object. Abstracting the descriptions away, a query answer in this application is of the form (Oid: Val), in which the first component denotes the identifier of an object and the second component is a *projection* upon the information content that users are interested in. The representation (Oid: Val) in an OWL ontology is called an *assertion*, thus, this class of object queries is called *assertion queries*.

Figure 1.1 elaborates a fairly general application for query answering over some information repository. In the context of this work, users can pose embedded SPARQL(-style) queries over OWL ontologies to retrieve objects and specify the details to be returned along with objects. The query answering mechanism is similar to conventional database technology in that a user query is translated into a chosen query plan for execution, except that query optimization is defined in this new context. Needless to say, the most difficult part in this application scenario is query performance. More specifically, the theme of the research is regarding the optimization of answering assertion queries over an OWL ontology (E1 and E3).

Finally, recall that relational DBMSs use parametrized SQL queries to save resources and enhance performance. The rational is to apply query optimization techniques to a class of queries instead of individual queries. Assertion queries also lend themselves nicely to parameterization; for instance, the query (1.1) can be parameterized for the price of cameras: the value "1000" can be replaced by a parameter. The substituted query, which now represents a *class* of queries, can be run by the query processor for query optimization.

1.1 Contributions

This dissertation concentrates on the introduction of a general form of *object queries*, called *assertion queries*, over OWL ontologies. A detailed list of the contributions follows, together with an illustration of the contributions in Figure 1.3.

1. Chapter 3 proposes assertion queries over knowledge bases in which it becomes possible for a user to control the format and content of additional facts about qualifying objects returned in response. By choosing a presentation of object data in the form of a description logics knowledge base, we provide a way to resolve the problem of incorporating ontological domain knowledge for object queries.

   Assertion queries are also a generalization of the well-known *instance queries* over description logics knowledge bases. Assertion queries are SPARQL-like, but its formal semantics has the capability to compute query results that resemble nested relational values and to deal with existential variables. Details about these particular features of assertion queries can be found in Section 3.3.

2. Chapter 4 presents an important technology for answering assertion queries, called *ABox absorption*. Since instance checking underlies assertion queries, ABox absorption is designed for scalable instance checking for the DL dialect $\mathcal{SHIQ}(\mathbb{D})$.

   ABox absorption is a non-trivial generalization of binary absorption. It operates with the assumption that a description logics knowledge base is consistent. In particular, ABox absorption allows instance checking tasks in the DL dialect $\mathcal{SHIQ}(\mathbb{D})$ to be mapped to concept subsumption tasks in the dialect $\mathcal{SHOIQ}(\mathbb{D})$, where a reasoner can usefully avoid reasoning about irrelevant ABox individuals and concrete facts. Since many existing optimizations for query answering cannot handle disjunctions, ABox absorption is particularly effective when disjunctions are necessary.

3. Chapter 5 presents the query compilation phase that underlies answering assertion queries, which exploits a plan language defined in Chapter 3. A particular note is that Chapter 5 shows how a combination of a knowledge base and a set of cached query results enables a transfer of basic database paradigms, for example, index scanning, view based query rewriting, among others. Cached query results, analogous to materialized views in relational databases, contain all the requested information about the qualifying objects. Cached query results can be leveraged to generate query plans that are more computationally favourable than the default plan.

   Chapter 5 also shows that, under some circumstances, it is possible to obtain query plans that avoid reasoning with respect to the background knowledge base. In this case, a query plan consists of operators that can be implemented without referencing any knowledge base, which, together with the use of cached query results, can result in a significant improvement over the query performance. Note that the aforementioned problem has not yet been addressed in the conventional view based query rewriting.

4. This dissertation also presents our implementation for answering assertion queries over knowledge bases, called *CARE Assertion Retrieval Engine* (Section 4.4). CARE is a Java-based system that implements the aforementioned ABox absorption and query optimization framework for assertion queries, including query rewriting and cost-based plan selection. The core component of CARE is a description logics reasoner for the DL dialect $\mathcal{SHI}(\mathbb{D})$. We also report the results of a number of empirical studies that validate the efficacy of the proposed ABox absorption algorithm in Section 4.4 and the benefit of query optimization in Section 5.4.

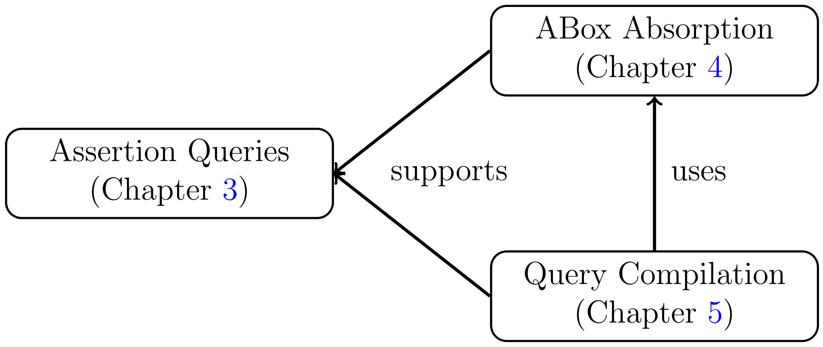


Figure 1.3: Contributions of the Dissertation.

1.2 Organization

This dissertation is organized as follows. Chapter 2 reviews the background work and related research. In Chapter 3, we define a user query language for assertion retrieval, in particular, projection descriptions that are specified by users to obtain details about objects are introduced. A list of procedures for computing projections is given in this chapter, followed by an introduction to the query plan language to be used in subsequent query compilation. Since instance checking underlies query answering in assertion retrieval, Chapter 4 focuses on the novel algorithm, ABox absorption, which maps instance checking tasks to concept satisfaction problems. A procedure for ABox absorption is also given in this chapter. In addition, Section 4.4 introduces an implementation for assertion retrieval, CARE, and presents the experimental results regarding the effectiveness of ABox absorption. Chapter 5 then follows to discuss query compilation of assertion queries. Specifically, this chapter elaborates a sequence of rewriting rules for plan generation and presents the implementation details of a cost-based plan selection strategy in CARE. The next and final chapter, Chapter 6, concludes this dissertation and proposes a list of problems to work on in the future.

Chapter 2

Background and Related Work

This work concerns query answering over knowledge bases, which, as an emerging topic, has raised a number of studies from different perspectives. A knowledge base is a special kind of database; however, there are some distinctive characteristics associated with query answering over knowledge bases. In relational databases a schema consists of various constraints, expressed in the form of *integrity constraints*. A schema is generally used to ensure such dependencies are satisfied in databases, while it is used only during query compilation but not during query answering. However, the schema in knowledge bases is always used in query answering, because knowledge may be implicit in knowledge bases and reasoning is therefore indispensable. The involvement of reasoning makes query answering and query optimization more challenging over KBs than in relational databases; in addition, the computational properties of query answering over KBs may not be as favourable as in relational databases.

There is another subtlety for query answering over knowledge bases. Recall that a logical theory $\mathbb{T}$ is *complete* if for every sentence ϕ in its language either $\mathbb{T} \models \phi$ or $\mathbb{T} \models \neg\phi$. Therefore, $\mathbb{T}$ is *incomplete* if there is a sentence ϕ in its language that both $\mathbb{T} \not\models \phi$ and $\mathbb{T} \not\models \neg\phi$ hold. Considering a database as a logical theory $\mathbb{T}$, it is defined using a finite specification in the form of a table. Typically, such a specification is assumed to be complete. To interpret missing facts from its representation table, the so-called *close-world assumption* (CWA) [Abiteboul et al., 1995] is adopted, which is in spirit to *negation as failure*. Negative information, by completeness, is thus logically valid in relational database.

The Web, being inherently open and incomplete, adopts the *open-world assumption* (OWA), which, in contrast to CWA, does not assume negative information a priori. The choice between CWA and OWA can have a dramatic impact on query answering over data repositories. Example 1 shows a case in which CWA and OWA affect the consistency of a repository and the query answers.

Example 1. Consider the following axioms in the data repository of some company selling cameras:

$$\textit{Every product is either a digital compact camera or a digital SLR camera.} \tag{2.1}$$

Consider CWA applied to relations and some object p in the repository that is explicitly declared to be a product. Because p is not declared to be either a digital compact camera or a digital SLR camera, the data repository would assume both (2.2) and (2.3) hold, which, however, makes the repository inconsistent because of (2.1).

$$p \textit{ is not a digital compact camera.} \tag{2.2}$$

$$p \textit{ is not a digital SLR camera.} \tag{2.3}$$

Under OWA, neither (2.2) nor (2.3) are assumed. Clearly, the two assumptions may result in different answers to the same query. Consider a user query that finds all products that are *not* digital SLR cameras. Then, p would be included in the query answers under CWA, but not so under OWA. □

The remainder of this chapter discusses several widely used approaches to Web data representation in Section 2.1, followed by an introduction to description logics (DL) in Section 2.2, which serves as the theoretic foundations of this work. Section 2.3 addresses query answering over knowledge bases; in particular, it reviews the state-of-the-art approaches to scalable conjunctive query answering over knowledge bases.

2.1 The Resource Description Framework

The Resource Description Framework (RDF) is generally used to represent information about Web resources (essentially objects), such as document titles, authors, among others, as well as about certain properties and values. RDF describes resources using statements that can be compactly encoded as triples $(subject, predicate, object)$. All three components in a triple are identified by an Internationalized Resource Identifier (IRI), which is a sequence of characters from the universal character set. In particular, URLs are a subset of IRIs. In the dissertation, qualified names (QNames) may be used to substitute full IRI references; for example, the QName `dbuw:object` is a shorthand for `http://db-tom.cs.uwaterloo.ca/object`, in which the prefix `dbuw:` is assigned to the namespace `http://db-tom.cs.uwaterloo.ca/`, followed by the local name `object`. In addition to the prefix `dbuw:` specifically introduced for examples in this dissertation and the `_:` prefix for indicating anonymous resources (called blank nodes in RDF datasets), the following conventional prefixes are used:

`rdf:` for the URI `http://www.w3.org/1999/02/22-rdf-syntax-ns#`,

`rdfs:` for the URI `http://www.w3.org/2000/01/rdf-schema#`,

`owl:` for the URI `http://www.w3.org/2002/07/owl#`, and

`xsd:` for the URI `http://www.w3.org/2001/XMLSchema#`.

Because RDF has triples as its underlying structure for expressions, any RDF expressions, as a collection of triples, can be described as an RDF graph, in which *subjects* and *objects* are nodes and *predicates* are edges. In normative RDF, there are restrictions imposed on what can be a subject, a predicate or an object. Formally, a RDF graph can be defined as follows:

Definition 1. ***RDF Graph.*** Let $\mathbf{I}, \mathbf{B}, \mathbf{L}$ denote disjoint infinite sets of IRIs, blank nodes, and literals, respectively. An RDF graph, $\mathbf{G}$, is a set of RDF triples, i.e., $\mathbf{G} = \{(\mathbf{s}, \mathbf{p}, \mathbf{o}) \mid (\mathbf{s}, \mathbf{p}, \mathbf{o}) \in (\mathbf{I} \cup \mathbf{B}) \times \mathbf{L} \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})\}$. □

Figure 2.1 shows a simple RDF graph, in which ***`rdf:type`*** is a predefined predicate in the RDF vocabulary that classifies objects into different types or classes, which means "an instance of," i.e., *ISA* relationship; for instance, the resource identified by `dbuw:Weddell` is an instance of the class `dbuw:Prof`. Also observe that `_:addr` is the so-called blank nodes, which denotes some anonymous resources. Note that RDF per se provides no mechanism for defining application-specific classes or properties [Manola and Miller, 2004].

RDF stores triples in a specific XML language, RDF/XML. It should be noted that XML models can serve the purpose of querying documents because they are flexible for transmitting structured information. The flexibility of representation, however, also leads to flexible queries. In contrast, RDF represents every statement in a triple and the actual RDF/XML representation is not relevant, so, the information content of statements is the concern of RDF, not the structural information; for example, the order of resources does not matter in RDF/XML due to the URI attached to them. As a consequence, querying RDF graph models is more straightforward than querying XML data models.

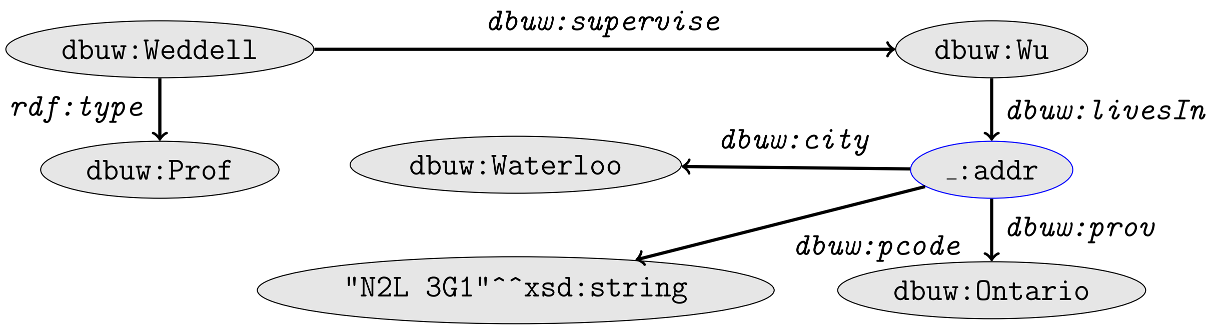


Figure 2.1: An example RDF graph.

2.1.1 Extensions to RDF

RDF has its own disadvantages and limitations. As discussed above, users may want to add application-specific classes while RDF provides no such facilities. In short, the expressiveness limits the usability of RDF. Consequently, the RDF Schema (RDFS) supplies such a mechanism by defining a set of RDF resources with special meanings, identified by the `rdfs:` prefix. RDFS thus does not add to the RDF vocabulary, but it interprets some built-in terms that must be processed by RDF applications to deal with RDFS. For example, RDFS allows for the addition of the following triple in the RDF graph in Figure 2.1 (`dbuw:Prof` *`rdf:type`* `rdfs:Class`), explicitly stating that `dbuw:Prof` is a class. RDFS also allows users to specify subclasses, subproperties, domain and range of properties, and so on. Readers can refer to [Brickley and Guha, 2004] for a complete list of RDFS features. While RDF data sets are always consistent because of the inability to specify contradictions, RDFS does allow contradictory (inconsistent) facts.

Although RDF Schema adds to RDF a list of features for enriching expressiveness, additional capabilities are useful and necessary as part of the development of the Semantic Web, for example, cardinality constraints on properties and the ability to describe new classes in terms of existing ones, among others. Further development of RDF/S becomes the targets of today's ontology languages, prominently OWL [Hitzler et al., 2012].

The Semantic Web, envisioned as an evolution of the existing World Wide Web and an infrastructure for knowledge exchange, gives rise to the popularity of ontologies. Formally an ontology, defined in [Gruber, 1993], is an "explicit specification of a conceptualization," designed for sharing resources and reusing components. OWL 2 is a declarative language for expressing ontologies. Specifically, an OWL ontology includes a terminology (vocabulary) describing domain knowledge and assertional knowledge that describes domain objects.

OWL 2 has a concrete syntax for ontology storage and exchange, based on RDF/XML. In this section only a fraction of the OWL 2 syntactic features is discussed. A thorough study about the syntax and semantics of OWL 2 is beyond the scope of this work, and interested readers can consult additional references, for example [Hitzler et al., 2012]. As an extension to RDF/S, OWL 2 also identifies all resources with an IRI.

In OWL 2, resources are called entities, associated with IRIs. Entities are the fundamental units in building an ontology. Entities fall into the following categories: classes (including datatypes), properties, and individuals. Classes represent sets of individuals; properties represent relationships in the domain; and individuals represent actual objects in the domain. Datatypes are analogous to classes, which refer to sets of data values. Similar to RDF, OWL 2 ontologies also have, in addition to entities, literals in different datatypes.

Individuals in OWL ontologies refer to objects in the domain. Analogous to blank nodes in RDF, OWL 2 also allows for anonymous nodes that are intended to be used only within an ontology. Properties are differentiated by the types of participants: object properties represent relationships between a pair of individuals, data properties represent relationships between an individual and a literal, and the built-in annotation properties are used to provide annotations.

Classes are application-specific, except for two OWL built-in classes `owl:Thing` and `owl:Nothing` with a predefined semantics such that the former represents the set of all individuals and the latter represents the empty set. Given a set of defined classes, composite classes can be built using general propositional operations, such as intersection, union, and complement. Imposing restriction on properties also constructs new classes. For instance, `ObjectSomeValuesFrom(dbuw:supervise dbuw:Person)` defines a class in which every instance supervises some person, given that `dbuw:supervise` is defined to be an object property and `dbuw:Person` a class; `DataHasValue(dbuw:pcode "N2L 3G1"^^xsd:string)` defines the class of instances (e.g., addresses) that have the postal code `N2L 3G1`. Datatype restrictions in `DataRange` allow one to limit the values of a datatype, which can be used to express more classes. The following class, for instance, is the set of instances in which each object has an age of at least 18:

```
DataAllValuesFrom(dbuw:age
    DatatypeRestriction(xsd:integer xsd:minInclusive "18"^^xsd:integer))
```

Entities are subsequently used to define axioms, which are the core of an ontology to express terminological knowledge (class axioms), declarations about properties (property axioms), and factual knowledge (assertion axiom) in the underlying domain. In particular, properties can be made complex by property axioms, for example, inverse and transitive.

Following the previous example, using OWL 2 class expression axioms, it is possible to state that the class `dbuw:Prof` is a subclass of the class in which instances supervise some person and that all adults in Ontario are exactly the set of persons with an age of at least 18. In addition, factual knowledge about actual objects can also be stated as axioms, for example, `dbuw:Weddell` is a professor, an adult in Ontario and is different from the object `dbuw:Wu`. In addition, it is assumed every object has at most one age, i.e., the property `dbuw:age` is declared to be functional by a property axiom. These statements are represented in OWL 2 syntax in Figure 2.2.

```
SubClassOf(dbuw:Prof ObjectSomeValuesFrom(dbuw:supervise dbuw:Person))
EquivalentClasses(dbuw:Adults DataAllValuesFrom(dbuw:age
     DatatypeRestriction(xsd:integer xsd:minInclusive "18"^^xsd:integer)))
ClassAssertion(dbuw:Weddell dbuw:Prof)
ClassAssertion(dbuw:Weddell dbuw:Adults)
DifferentIndividuals(dbuw:Weddell dbuw:Wu)
FunctionalDataProperty(dbuw:age)
```

Figure 2.2: Example OWL 2 axioms.

The expressiveness of OWL 2 can be observed through the examples, especially when compared to RDF/S. System modelling often requires a large number of complicated constraints to be imposed in real applications, and OWL 2 offers the appropriate mechanism for ontology engineers to do so. The expressiveness of OWL 2 inevitably makes reasoning over OWL ontology computationally expensive. The next section discusses the computational properties of OWL 2 and some of its sublanguages, called OWL 2 profiles, that trade expressiveness for tractability. Finally, it should be noted that there are types of statements that cannot be expressed in OWL 2. As an example, OWL 2 does not provide constructs to compare functional data properties (regarded as attributes or features). Therefore, an attempt to model the statement “products that have current prices lower than the manufacturer suggested prices” would fail in OWL 2.

OWL 2 Profiles

To consider the computational complexity of OWL ontologies, there are several options in terms of measures. Particularly, one measure is to consider the complexity of the class axioms in the ontology, this is the *taxonomic (schema) complexity* measure [Motik et al., 2012]. When queries are posed over ontologies, *data complexity* measured with respect to the size of the assertion axioms in the ontology is of particular interest. OWL 2 corresponds to the description logic $\mathcal{SROIQ}(\mathbb{D})$ (see Section 2.2) in terms of model-theoretic semantics, and taxonomic complexity of OWL 2 is N2EXPTIME-complete. However, when the semantics of OWL 2 ontologies is defined as an extension of RDF semantics, taxonomic complexity of OWL 2 is undecidable. Data complexity of OWL 2 is known to be decidable in model-theoretic semantics, yet it is undecidable in RDF semantics.

Some domain applications, however, do not take advantage of the expressive power of the full OWL 2. For these applications using the full OWL 2 becomes awkward because of the undesirable computational properties. To deal with such issues, OWL 2 profiles [Motik et al., 2012], which are sublanguages of the full OWL 2, are introduced to trade some expressive power for the efficiency of reasoning. Three profiles of OWL 2 are available in OWL 2 for different application scenarios: OWL 2 EL, OWL 2 QL and OWL 2 RL.

OWL 2 EL has its name from the well-known description logic family $\mathcal{EL}$ (cf. Definition 6), of which both taxonomic and data complexity are in PTime-complete. This profile disallows the use of some class constructs that are inherently difficult to deal with, including negations, disjunctions, universal and cardinality restrictions, inverse properties, among others. As an example, the axioms in Figure 2.2 can be fully supported in this profile. The intended use of OWL 2 EL profiles is when a large volume of classes and properties are present in the ontology.

The second profile, OWL 2 QL, is designed for querying, i.e., when massive instance data (actual objects) are used in an ontology. In these scenarios data is stored in relational databases and query answering is the central reasoning task; furthermore, analogous to evaluating SQL over conventional relational databases, OWL 2 QL enables efficient query answering, of which data complexity is AC^0 and taxonomic complexity is NLogSpace-complete. This profile is based on the *DL-Lite* family of description logics (cf. Definition 7), and possesses essential features needed to express UML and ER diagrams. Particularly, functional properties and existential restrictions over some class are not supported in this profile, which means the example in Figure 2.2 is beyond the expressiveness of OWL 2 QL. Indeed, this profile imposes more restrictions than OWL 2 EL on the expressiveness in order to achieve the query performance comparable to that of relational databases.

The last profile, OWL 2 RL, considers applications that require scalable reasoning as well as sufficient expressiveness. Reasoning with this profile can be realized by rule-based implementation, which has nice computational properties, i.e., PTime-complete for both taxonomic and data complexity. OWL 2 RL limits the positions where OWL 2 syntactic artifacts can appear, thus restricting the expressiveness. For instance, existential restrictions cannot occur on the right-hand side of class axioms, which renders the first axiom in Figure 2.2 inexpressible in this profile.

To formally discuss the profile OWL 2 QL , the theoretic underpinnings, i.e., the family of *DL-Lite* of description logics, will be elaborated in later sections. As DLs play a fundamental role in OWL 2 and its profiles, the next section addresses the syntax and semantics of relevant description logics, as well as core reasoning tasks.

2.2 Description Logics

Description logics (DLs) represent a family of logic fragments, of which most are decidable sub-logic of first-order logic (FOL) [Baader et al., 2003, Chapter 1]. Instead of introducing the syntax and semantics of the full OWL 2, several dialects of DLs with diverse expressiveness and computational properties are introduced. First, a list of DL concept constructs are introduced in Definition 3; in particular, datatypes, i.e., the *concrete domains* [Baader et al., 2003; Lutz, 2003], is given in Definition 2. Next, the semantics of these constructs are defined. Finally, several DL dialects that are referred to in this work are defined.

Definition 2. ***Concrete Domain***. A concrete domain is a pair $(\Delta_{\mathbb{D}}, \Phi_{\mathbb{D}})$, where $\Delta_{\mathbb{D}}$ is a set of concrete values and $\Phi_{\mathbb{D}}$ is a set of predicate names. A predicate name P is associated with an arity n and an n-ary predicate, interpreted as $P^{\mathbb{D}} \subseteq \Delta_{\mathbb{D}}^n$.

The concrete domain discussed in this work is the *string* domain, where $\Delta_{\mathbb{D}} = \mathbb{D}$ is the set of all finite strings and $\Phi_{\mathbb{D}}$ consists of the following predicates:

- unary predicates $=_s$ with interpretation, for each $s \in \mathbb{D}$, $(=_s)^{\mathbb{D}} = \{s' \in \mathbb{D} \mid s' = s\}$;
- binary predicate $<$ with interpretation $(<)^{\mathbb{D}} = \{(s, s') \mid \{s, s'\} \subseteq \mathbb{D} \text{ and } s < s'\}$.

$\square$

The predicates that can be defined in a concrete domain are not limited to $=$ and $<$, and it is legitimate to, for example, define $<$ as unary predicates $<_s$, $=$ a binary predicate as well. However, the above two predicates suffice to define the concrete domain concepts used in this work, with appropriate syntactic abbreviations introduced later.

Definition 3. ***Concept Constructs***. Let $\{A, A_1, \ldots\}$, $\{R, R_1, \ldots\}$, $\{f, f_1, \ldots, f_n, g, g_1, \ldots, g_n\}$ and $\{a, a_1, a_2, \ldots\}$ denote countably infinite and disjoint sets of *concept names* NC, *role names* NR, *concrete features* NF and *individual names* NI, respectively.

A *role* is defined as:

$$\begin{aligned} S ::= \quad & R & \text{(atomic role)} \\ \mid \quad & R^- & \text{(inverse role)} \end{aligned}$$

and the inverse of a role S is defined as follows to avoid considering roles such as S^{--}:

$$S^- = \begin{cases} R^- & \text{if } S = R \\ R & \text{if } S = R^-. \end{cases}$$

A role constraint $\mathcal{C}_R$ is defined over some roles, which can be one of the following:

$$\begin{aligned} \mathcal{C}_R ::= \quad & S_1 \sqsubseteq S_2 & \text{(role inclusion)} \\ \mid \quad & \texttt{Trans}(S) & \text{(transitive role)} \end{aligned}$$

A role hierarchy, denoted $\mathcal{R}$, is a finite set of role constraints in the form of $S_1 \sqsubseteq S_2$ or $\texttt{Trans}(R)$. Note that $S_1 \sqsubseteq S_2$ implies $S_1^- \sqsubseteq S_2^-$. Let $\overset{*}{\sqsubseteq}_{\mathcal{R}}$ be the transitive-reflexive closure of $\sqsubseteq$ over the set $\mathcal{R} \cup \{S_1^- \sqsubseteq S_2^- \mid S_1 \sqsubseteq S_2 \in \mathcal{R}\}$. When the role hierarchy is clear from the context, we write $\overset{*}{\sqsubseteq}$ instead. Two roles are considered equivalent w.r.t. the role hierarchy $\mathcal{R}$, denoted $S_1 \equiv_{\mathcal{R}} S_2$, if $S_1 \overset{*}{\sqsubseteq}_{\mathcal{R}} S_2$ and $S_2 \overset{*}{\sqsubseteq}_{\mathcal{R}} S_1$. Furthermore, because a role S is transitive iff S^- is transitive, we define $\texttt{Trans}(S) \in \mathcal{R}$ iff if there is $R \equiv_{\mathcal{R}} S, \texttt{Trans}(R) \in \mathcal{R}$ or $\texttt{Trans}(R^-) \in \mathcal{R}$. A role S is called *simple* w.r.t. $\mathcal{R}$ if $\texttt{Trans}(S') \notin \mathcal{R}$ for all $S' \overset{*}{\sqsubseteq}_{\mathcal{R}} S$.

A *concept* is defined as follows:

$$\begin{aligned} C, D ::= \quad & \top & \text{(top)} \\ \mid \quad & \bot & \text{(bottom)} \\ \mid \quad & A & \text{(atomic concept)} \\ \mid \quad & \neg C & \text{(negation)} \\ \mid \quad & C \sqcap D & \text{(conjunction)} \\ \mid \quad & \exists S.C & \text{(existential quantification)} \end{aligned}$$

$$
\begin{array}{lll}
\mid & \exists g_1, \ldots, g_n.P & \text{(domain concept)} \\
\mid & \exists^{\leq n} S.C \text{ for some } \textit{simple } S & \text{(at-most qualified number restriction)} \\
\mid & \{a\} & \text{(nominal)}
\end{array}
$$

where n is a non-negative integer, and P is a predicate of arity n. Finally, a constraint $\mathcal{C}$ is defined as follows:

$$
\begin{array}{rlll}
\mathcal{C} ::= & C \sqsubseteq D & \text{(concept inclusion)} \\
\mid & a_i : C & \text{(concept assertion)} \\
\mid & S(a_i, a_j) & \text{(role assertion)} \\
\mid & \mathcal{C}_R & \text{(role constraint).}
\end{array}
$$

A knowledge base $\mathcal{K}$ is a pair $(\mathcal{T}, \mathcal{A})$, where $\mathcal{T} = \{C \sqsubseteq D\} \cup \{\mathcal{C}_R\}$ and $\mathcal{A} = \{a_i : C\} \cup \{S(a_i, a_j)\}$. □

Several syntactic abbreviations are used in this work. Specifically, an *equivalence* $C \equiv D$ is used in place of both $(C \sqsubseteq D)$ and $(D \sqsubseteq C)$, a *disjunction* $C \sqcup D$ in place of $\neg(\neg C \sqcap \neg D)$, and a *universal quantification* $\forall S.C$ in place of $\neg \exists S.(\neg C)$. Note that only *simple* roles are allowed in at-most qualified number restrictions to yield a decidable dialect [Horrocks et al., 1999]. In addition, some special cases of at-most qualified number restrictions deserve attention. At-least qualified number restrictions $\exists^{\geq n} S.C$ are considered syntactic variants of $\neg(\exists^{\leq n-1} S.C)$, and at-most *number restrictions*, if used in any logic, is synonymous to $\exists^{\leq n} S.\top$. $\forall S.C$ (resp. $\exists S.C$) are considered to be shorthand for the concept $\exists^{\leq 0} S.\neg C$ (resp. $\exists^{\geq 1} S.C$).

Domain concepts may also be written in a more straightforward way. Particularly, a *feature equality* $g = s$, for a constant string s, simplifies $\exists g.(=_s)$, a *feature linear order* $g_1 < g_2$ simplifies $\exists g_1, g_2. <$, and a concept in the form of $g_1 \leq s$ is an abbreviation for $(g_1 = s) \sqcup ((g_1 < g_2) \sqcap (g_2 = s))$. Also, we might use $(t_1 \texttt{ op } t_2)$ to generalize concrete domain concepts such that t_1 and t_2 are either a concrete feature or a finite string and $\texttt{op} \in \{<, =\}$. Section 2.1.1 notes that OWL 2 does not provide facilities for comparing functional data properties (corresponding to concrete features in Definition 3), yet such a class construct, i.e., domain concept, is available in Definition 3.

The TBox and ABox of a knowledge base express intensional and extensional knowledge, respectively. A TBox $\mathcal{T}$ is called *primitive* iff it consists entirely of axioms of the form $A \equiv C$ with $A \in \text{NC}$, each $A \in \text{NC}$ appears in at most one left hand side of an axiom, and $\mathcal{T}$ is *acyclic*. Acyclicity is defined as follows: $A_1 \in \text{NC}$ *directly uses* $A_2 \in \text{NC}$ if

$A_1 \equiv C \in \mathcal{T}$ and A_2 occurs in C; "uses" is the transitive closure of "directly uses". Then $\mathcal{T}$ is *acyclic* if there is no $A \in$ NC that uses itself. $A \in$ NC *is defined in* $\mathcal{T}$ if $\mathcal{T}$ contains $A \sqsubseteq C$ or $A \equiv C$.

Definition 4. ***Semantics***. The semantics of the constructs in Definition 3 is defined in the standard Tarski-style. An interpretation $\mathcal{I}$ is a 2-tuple $(\triangle \uplus \Delta_{\mathbb{D}}, (\cdot)^{\mathcal{I}})$ in which $\triangle$ is a non-empty *abstract domain* of objects, $\Delta_{\mathbb{D}}$ is a *concrete domain* of objects and $(\cdot)^{\mathcal{I}}$ is an interpretation function. The interpretation function $\cdot^{\mathcal{I}}$ maps each concept name A to a set $(A)^{\mathcal{I}} \subseteq \triangle$, each role name R to a relation $(R)^{\mathcal{I}} \subseteq (\triangle \times \triangle)$, each concrete feature g to a total function $g^{\mathcal{I}}\colon \triangle \rightarrow \Delta_{\mathbb{D}}$, and each individual a to a domain element $o \in \triangle$. The interpretation is extended to inverse roles as follows: $(R^{-})^{\mathcal{I}} = \{(o_2, o_1) \in \triangle \times \triangle \mid (o_1, o_2) \in R^{\mathcal{I}}\}$.

The interpretation function is extended to all concepts in the following way:

$$
\begin{array}{lcl}
(\top)^{\mathcal{I}} & = & \triangle \\
(\bot)^{\mathcal{I}} & = & \emptyset \\
(\neg C)^{\mathcal{I}} & = & \triangle \setminus (C)^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} & = & (C)^{\mathcal{I}} \cap (D)^{\mathcal{I}} \\
(\exists S.C)^{\mathcal{I}} & = & \{o_1 \in \triangle \mid \exists o_2 \in (C)^{\mathcal{I}} : (o_1, o_2) \in (S)^{\mathcal{I}}\} \\
(\exists g_1, \ldots, g_n.P)^{\mathcal{I}} & = & \{o \in \triangle \mid \exists s_1, \ldots, s_n \in \Delta_{\mathbb{D}} : {g_i}^{\mathcal{I}}(o) = s_i, \text{ for } 1 \leq i \leq n \\
 & & \qquad \text{and } (s_1, \ldots, s_n) \in P^{\mathbb{D}}\} \\
(\exists^{\leq n} S.C)^{\mathcal{I}} & = & \{o_1 \in \triangle \mid \#\{o_2 \in (C)^{\mathcal{I}} \mid (o_1, o_2) \in (S)^{\mathcal{I}}\} \leq n\} \\
(\{a\})^{\mathcal{I}} & = & \{(a)^{\mathcal{I}}\}
\end{array}
$$

where $\#X$ denotes the cardinality of a set X. The satisfaction relation for a constraint $\mathcal{C}$ is defined in the usual way:

$$
\begin{array}{lll}
\mathcal{I} \models C \sqsubseteq D & \text{iff} & (C)^{\mathcal{I}} \subseteq (D)^{\mathcal{I}} \\
\mathcal{I} \models a_i : C & \text{iff} & (a_i)^{\mathcal{I}} \in (C)^{\mathcal{I}} \\
\mathcal{I} \models S(a_i, a_j) & \text{iff} & ((a_i)^{\mathcal{I}}, (a_j)^{\mathcal{I}}) \in (S)^{\mathcal{I}} \\
\mathcal{I} \models S_1 \sqsubseteq S_2 & \text{iff} & (S_1)^{\mathcal{I}} \subseteq (S_2)^{\mathcal{I}} \\
\mathcal{I} \models \texttt{Trans}(S) & \text{iff} & (o_1, o_3) \in (S)^{\mathcal{I}} \text{ whenever } (o_1, o_2) \in (S)^{\mathcal{I}} \text{ and } (o_2, o_3) \in (S)^{\mathcal{I}} \\
 & & \quad \text{for } o_i \in \triangle, i \in \{1, 2, 3\}
\end{array}
$$

The satisfaction relation also extends to a knowledge base $\mathcal{K}$ such that $\mathcal{I} \models \mathcal{K}$ iff $\mathcal{K}$ satisfies all the constraints in $\mathcal{K}$. In this case, the interpretation $\mathcal{I}$ is called a model of $\mathcal{K}$

(and of the appropriate $\mathcal{T}$ and $\mathcal{A}$). We write $\mathcal{K} \models \mathcal{C}$ if all interpretations that satisfy $\mathcal{K}$ also satisfy the constraint $\mathcal{C}$. $\square$

Now we are ready to introduce several DL dialects. The naming convention for DLs uses the following name constituents:

$$(\mathcal{ALC} \mid \mathcal{S})\ [\mathcal{H}]\ [\mathcal{O}]\ [\mathcal{I}]\ [\mathcal{Q}]\ [(\mathbb{D})]$$

These name constituents are explained below:

- $\mathcal{ALC}$, an abbreviation for *attributive language with complements*, allows concepts formed by the first six concept constructs in Definition 3, with the limitation that an existential quantification must be over *atomic roles* (i.e., $\exists R.C$).
- $\mathcal{S}$ is $\mathcal{ALC}$ with the role constraint $\texttt{Trans}(S)$.
- $\mathcal{H}$ denotes the role constraint $S_1 \sqsubseteq S_2$.
- $\mathcal{O}$ denotes nominals.
- $\mathcal{I}$ denotes inverse roles.
- $\mathcal{Q}$ denotes at-most qualified number restrictions.
- $(\mathbb{D})$ denotes concrete domains.

A variety of DL dialects can then be named by concatenating the name constituents, for example, $\mathcal{ALCI}$, $\mathcal{SHI}(\mathbb{D})$, among others. Particularly, concrete domains in Definition 2 are useful for describing objects that involve concrete qualities, such as the price of a product and the name of a person. With all the concept constructs introduced in Definitions 2 and 3, the expressive dialect $\mathcal{SHOIQ}(\mathbb{D})$ is obtained.

In addition to $\mathcal{ALC}$ family dialects, two more DL dialects are given below: one is called $\mathcal{EL}$ [Baader, 2003] and the other represents an important family of DLs,the *DL-Lite* family, for data management. Before delving into the details of these dialects, the notion of *unique name assumption* is defined below.

Definition 5. ***Unique Name Assumption (UNA)***. UNA fixes distinct individual names a to be distinct elements of $\triangle$, i.e., $(a_i)^{\mathcal{I}} \neq (a_j)^{\mathcal{I}}$ if $i \neq j$. $\square$

The choice of adopting UNA impacts the computational properties of some DLs. OWL does not require UNA because there are axioms in OWL that can be used to explicitly state whether two individuals are the same or not, i.e. the `SameIndividual` and `DifferentIndividuals` axioms.

Definition 6. ***The DL dialect*** $\mathcal{EL}$. An $\mathcal{EL}$ concept is formed by the concept constructs *top*, *atomic concept*, *conjunction*, and *existential quantification*, with the limitation that an existential quantification must be over *atomic roles* (i.e., $\exists R.C$). □

Definition 7. ***DL-Lite***$^{\mathcal{H}}_{horn}$. The concept constructs of *DL-Lite*$^{\mathcal{H}}_{horn}$ include *bottom*, *atomic concept*, *negation*, and *existential restriction*. Concept inclusions are limited to $C^b_1 \sqsubseteq C^b_2$ and $C^b_1 \sqsubseteq \neg C^b_2$, where C^b_i can be concepts built from *bottom*, *atomic concept*, or *existential restriction*; role constraints are restricted to the form of $S_1 \sqsubseteq S_2$; concept assertions are limited to $a_i : A$ or $a_i : \neg A$; role assertions are allowed in the form of $S(a_i, a_j)$ or $\neg S(a_i, a_j)$. □

The logic *DL-Lite*$^{\mathcal{H}}_{horn}$ is the theoretic underpinning of the profile OWL 2 QL [Artale et al., 2009] discussed in Section 2.1.1. Although the *DL-Lite* family in general adopts UNA, the complexity of *DL-Lite*$^{\mathcal{H}}_{horn}$ is independent of UNA, i.e., the data complexity of *DL-Lite*$^{\mathcal{H}}_{horn}$ remains AC^0 for query answering. It should be noted that role constraints in Definition 3 can be supported in any *DL-Lite* logic, without affecting the complexity results. So, *DL-Lite*$^{\mathcal{H}}_{horn}$, together with role constraints can accommodate almost all constructs in the OWL 2 QL profile. A missing feature in *DL-Lite*$^{\mathcal{H}}_{horn}$ is the use of $a_i \approx a_j$ with the interpretation $(a_i)^{\mathcal{I}} = (a_j)^{\mathcal{I}}$. This construct is useful only if UNA is not adopted, as in OWL. It, however, affects data complexity; for instance, data complexity increases from AC^0 to LogSpace-complete [Artale et al., 2009]. The change in complexity resulting from individual equality is consequential for efficient query answering, as can be seen in Section 2.3.3.

Note that keys, which are constructs used to identify objects uniquely, are not expressible in any DL dialects defined previously, yet OWL 2 allows the use of keys. To introduce keys in knowledge bases, the $\mathcal{DLF}$ family [Toman and Weddell, 2005] has been studied by viewing keys as a generalization of functional dependencies. The $\mathcal{DLF}$ dialects focus on certain database features, especially functional dependencies; for example, the logic $\mathcal{CFD}$ [Khizder et al., 2000] in $\mathcal{DLF}$, short for "Classic Functional Dependency," is a dialect capable of describing object relational database schemas with uniqueness constraints and runs in PTime for subsumption tests. Since these dialects are beyond the scope of this work, interested readers can refer to the relevant literature.

Knowledge acquisition is usually obtained via reasoning. The next definitions [Baader et al., 2003, Chap. 2] consider some reasoning tasks that may be frequently referred to throughout this work.

Definition 8. ***Satisfiability (Consistency)***. A knowledge base $\mathcal{K}$ is *satisfiable* (or *consistent*) if there is a model of $\mathcal{K}$. □

Definition 9. ***Subsumption Checking***. A concept C is *subsumed* by a concept D with respect to $\mathcal{K}$, denoted $\mathcal{K} \models C \sqsubseteq D$, if $C^\mathcal{I} \subseteq D^\mathcal{I}$ holds for *every* model of $\mathcal{K}$. □

Consistency of a KB is the minimal requirement for reasoning with a KB. Analogously, an ABox $\mathcal{A}$ is called *consistent* with respect to $\mathcal{K}$, if there is a model of $\mathcal{K}$ that is also a model of $\mathcal{A}$. Many other reasoning tasks can be reduced to KB consistency. For instance, the subsumption check of $C \sqsubseteq D$ amounts to checking the consistency of the new KB $\mathcal{K}' = \mathcal{K} \cup \{a_0 : (C \sqcap \neg D)\}$, where a_0 is a fresh individual name that does not occur in $\mathcal{K}$. Therefore, $C \sqsubseteq D$ w.r.t. $\mathcal{K}$ holds iff $\mathcal{K}'$ is *not* consistent.

The notion of satisfiability can be extended to a concept C such that C is *satisfiable* with respect to $\mathcal{K}$, denoted $\mathcal{K} \models C$, if there is an interpretation $\mathcal{I}$ of $\mathcal{K}$ such that $C^\mathcal{I} \neq \emptyset$. The satisfiability of C is also reducible to the consistency of $\mathcal{K}$ because a concept C is *unsatisfiable* if $\mathcal{K} \models C \sqsubseteq \bot$.

Definition 10. ***Instance Checking***. An instance checking problem, denoted $\mathcal{K} \models (a : C)$, is to decide if a concept assertion in the form of $(a : C)$ is *entailed* by $\mathcal{K}$, i.e., it holds iff every model of $\mathcal{K}$ is also a model of $(a : C)$, i.e., $(a)^\mathcal{I} \in (C)^\mathcal{I}$ for every model $\mathcal{I}$ of $\mathcal{K}$. □

A central problem studied in this work is the *instance retrieval* reasoning problem (or simply *instance queries*): for a concept C, retrieve all individuals a in $\mathcal{K}$ such that $\mathcal{K} \models (a : C)$. Instance retrieval can be reduced to instance checking and one instance retrieval task often requires more than one instance checking; indeed, one such task, for a naïve implementation, will issue the same number of instance checking requests as that of individuals occurring in $\mathcal{K}$. Such an implementation for instance retrieval is also called *linear instance retrieval* [Haarslev and Möller, 2008].

Instance checking is also reducible to the consistency of $\mathcal{K}$, i.e., $\mathcal{K} \models (a : C)$ holds iff $\mathcal{K}' = (\mathcal{T} \cup \{A \sqsubseteq \neg C), \mathcal{A} \cup \{a_1 : A\}\}$ is *inconsistent*, where A is a fresh atomic concept and a_1 a fresh individual name. Conversely, the consistency of $\mathcal{K}$ can be reduced to instance checking as well, i.e., $\mathcal{K}$ is consistent iff $\mathcal{K} \not\models (a_1 : A)$ for A a fresh atomic concept and a_1 a fresh individual name.

Tableau Algorithm

A core reasoning task of DL reasoners, as shown in the previous section, is to decide the consistency (satisfiability) of a knowledge base $\mathcal{K}$. The algorithm adopted by DL reasoners to reason about the (in)consistency of $\mathcal{K}$ is the *tableau* algorithm [Baader et al., 2003]. The basic idea of this algorithm is to prove consistency by demonstrating the existence of a model $\mathcal{I}$ of $\mathcal{K}$. It works on a completion graph (usually a labeled graph), where a node x of the graph is labelled with a set of concepts $\mathcal{L}(x)$ and an edge is labelled with a set of roles S, e.g., $\mathcal{L}(\langle x_i, x_j\rangle) = \{S\}$ for two nodes x_i and x_j. Initially, a node x_i is created for every individual a_i in $\mathcal{K} = \{\mathcal{T}, \mathcal{A}\}$. For any constraint $a_i : C \in \mathcal{A}$ and $S(a_i, a_j) \in \mathcal{A}$, the algorithm defines $\mathcal{L}(x_i) = \{C, \{a_i\}\}$, $\mathcal{L}(x_j) = \{\{a_j\}\}$, and $\mathcal{L}(\langle x_i, x_j\rangle) = \{S\}$, respectively. Then, the completion graph is expanded by applications of expansion rules in Table 2.3, until the graph is *complete*. A completion graph is not, but closely resembles, a model of $\mathcal{K}$; for instance, a model can be infinite (though finitely representable), while a corresponding completion graph is not.

A completion graph is called *complete* when either a *clash* is encountered or no more tableau rules are applicable. In the latter case, the completion graph is clash-free, and a model of $\mathcal{K}$ is guaranteed to exist. For the DL dialect $\mathcal{ALCI}(\mathbb{D})$, a completion graph contains a clash if, for some node x,

1. $\bot \in \mathcal{L}(x)$, or
2. for some $A \in \mathrm{NC}$, $\{A, \neg A\} \subseteq \mathcal{L}(x)$, or
3. for some $f \in \mathrm{NF}$, $f < f \in \mathcal{L}(x)$, or
4. for some $f \in \mathrm{NF}$, $\{s, s'\} \subseteq \mathbb{D}$, $\{f = s, f = s'\} \subseteq \mathcal{L}(x)$ and $s \neq s'$.

Figure 2.3 defines a set of tableau rules for the DL $\mathcal{ALCI}(\mathbb{D})$. For some more expressive logic, additional expansion rules are necessary. Particularly, the $\sqcup$-rule is *non-deterministic* in the sense that a tableau has to nondeterministically choose C_1 or C_2. A choice, e.g., C_1, leads to a replication of the original completion graph, which is expanded further. If a clash is found in the replication, the tableau algorithm must backtrack to the original graph, and select C_2 to proceed.

The $\exists$-rule is a generating rule in that fresh nodes are introduced in completion graphs after applying this rule. It is possible that a sequence of nodes and edges of a completion graph repeats itself; therefore, *blocking* is defined to ensure termination of the algorithm so that a tableau will not generate an infinite number of nodes, for instance, a tableau

$\sqcap$-rule

Conditions:	$C_1 \sqcap C_2 \in \mathcal{L}(x)$ and $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$
Actions:	$\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1, C_2\}$

$\sqcup$-rule

Conditions:	$C_1 \sqcup C_2 \in \mathcal{L}(x)$ and $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$
Actions:	$\mathcal{L}(x) = \mathcal{L}(x) \cup \{\ C\}$ for some $C \in \{C_1, C_2\}$

$\exists$-rule

Conditions:	$\exists S.C \in \mathcal{L}(x)$ and there is *no* individual c s.t. $S \in \mathcal{L}(\langle a, c\rangle)$ and $C \in \mathcal{L}(c)$
Actions:	create a new node b with $\mathcal{L}(\langle a, b\rangle) = \{S\}$ and $\mathcal{L}(b) = \{C\}$

$\forall$-rule

Conditions:	$\forall S.C \in \mathcal{L}(x)$ and there is an individual b s.t. $S \in \mathcal{L}(\langle a, b\rangle)$ and $C \notin \mathcal{L}(b)$
Actions:	$\mathcal{L}(b) = \mathcal{L}(b) \cup \{C\}$

$\sqsubseteq$-rule

Conditions:	$C_1 \sqsubseteq C_2 \in \mathcal{K}$ and $\{\neg C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$
Actions:	$\mathcal{L}(x) = \mathcal{L}(x) \cup \{\ C\}$ for some $C \in \{\neg C_1, C_2\}$

Figure 2.3: Tableau rules for the DL dialect $\mathcal{ALCI}(\mathbb{D})$.

algorithm will not apply generating rules to nodes that are blocked. A number of blocking techniques have been developed, for instance, the one presented in [Horrocks and Sattler, 2002] suffices for $\mathcal{ALCI}(\mathbb{D})$.

The $\sqsubseteq$-rule deals with concept inclusions (a.k.a. axioms) by converting it into a disjunction and acts as the $\sqcup$-rule. Obviously, applications of the $\sqsubseteq$-rule degrade the reasoning performance because every axiom causes a disjunction to be added to *every* node in the completion graph. Given an axiom in $\mathcal{K}$ and a completion graph of n nodes for checking the consistency of $\mathcal{K}$, there could be as many as 2^n replications of the completion graph.

It has been known for some time in the case of of the $\sqsubseteq$-rule that *lazy unfolding* is an important optimization technique in model building algorithms [Baader et al., 1994]. It is also imperative for a large TBox, i.e., a set of axioms, to be manipulated by an *absorption* procedure to maximize the benefits of lazy unfolding in such algorithms, thereby reducing the combinatorial effects of disjunction in underlying tableaux procedures [Horrocks, 1998], such as the previously shown 2^n case. Absorption aims at transforming *general axioms* into ones that can be exploited by lazy unfolding. The basic absorption allows a rewriting of axioms into the form of $A_1 \sqsubseteq C$ and the form of $\neg A_2 \sqsubseteq C$, where both A_1 and A_2 are atomic concepts. These absorbed axioms are thus examples of unary absorption in that the left-hand side has only a single atomic concept. Hudek and Weddell [2006] extends the above absorption to rewrite axioms into the form of $A_1 \sqcap A_2 \sqsubseteq C$, called binary absorption. Furthermore, axioms of the form $\exists R.\top \sqsubseteq C$ could be absorbed by role absorption [Tsarkov and Horrocks, 2004]. To our knowledge, [Motik et al., 2009; Wu and Haarslev, 2008] are among the first to provide an absorption framework, instead of using any single absorption techniques presented above. In this work binary absorption is employed to absorb ABox data into TBox axioms, as further discussed in Chapter 4.

2.3 Query Answering over Knowledge Bases

Similar to databases, knowledge bases allow users to pose queries and to maintain the contents. Typically, knowledge can be updated or revised. In the former case old facts are replaced by new facts to reflect present situations, and the latter concerns the elimination of inconsistency when old facts conflict with new facts. Knowledge updates or revisions are nevertheless beyond the scope of this work. Instead, query answering over knowledge bases is our focus.

To answer queries over knowledge bases, a query must be expressed by some language. In relational databases, the standard surface query language is SQL. Similarly, there is a query language acting as SQL over RDF (and its extensions) data sets: the SPARQL query language. Section 2.3.1 gives a short introduction to this query language.

Since SQL queries are applications (with extensions) of relational algebra, a form of first-order queries, it is more convenient to use first-order queries to express these queries in the form of $\phi(\overline{x})$, where ϕ is a well-formed formula in first-order logic. Rather than dealing with full first-order queries, research in the area of query answering has primarily concentrated on *conjunctive queries*, a subset of first-order queries. Section 2.3.2 defines conjunctive queries and presents the state-of-the-art techniques for answering conjunctive queries over knowledge bases.

2.3.1 The SPARQL Query Language

The SPARQL query language [Prud'hommeaux and Seaborne, 2008] is the standard structural query language over RDF data. From a database perspective, SPARQL corresponds to SQL over relational databases. SPARQL queries are usually evaluated on the basic building blocks. Once an output pattern is computed, the solution modifiers in the queries modify the results, for instance, projection, order, limit, among others. Figure 2.4 show an example SPARQL query, which is duplicated from the query in Figure 1.1.

```
SELECT    ?x  ?cn  ?p
WHERE {
        ?x type Digital_SLR .
        ?x name ?cn .
        ?x release ?d .  FILTER (?d > 20100101) .
        ?x price ?p .  FILTER (?p < 1000)
     }
```

Figure 2.4: An example SPARQL query.

This section does not attempt to outline the complete details of SPARQL, which are available in [Prud'hommeaux and Seaborne, 2008; Pérez et al., 2009]. As mentioned earlier, SPARQL query evaluation is based on obtaining results from *basic graph patterns* (BGPs). Additional operations can then be performed on BGPs to obtain more expressive patterns, for example, `AND`, `UNION`, and `OPTIONAL`. This section gives an informal discussion on evaluating BGPs.

Definition 11. ***Basic Graph Pattern***. Let $\mathbf{I}$, $\mathbf{L}$, and $\mathbf{V}$ denote sets of IRIs, literals, and variables, respectively. A SPARQL graph pattern expression is defined as follows:

- a tuple $\mathbf{t} \in (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$ is a triple graph pattern, where $\mathbf{V}$ denotes the set of variables, and

- for a graph pattern e and a SPARQL boolean condition c, (e `FILTER` c) is also a graph pattern.

A basic graph pattern is then a set of triple graph patterns, each of which can optionally have `FILTER` operations. □

Intuitively, the example query in Figure 2.4 finds digital SLR cameras that have a price less than 1000 and that are released after a given date. For data captured by RDF triples in the form of (*subject*, *predicate*, *object*), particular choices for *predicate* in OWL 2 have added consequences: existing triples that include such choices can mandate the inference of additional triples. In OWL 2, this flexibility is an option determined by choosing an appropriate *entailment regime* [Glimm and Ogbuji, 2013]. Different entailment regimes allow for different uses of semantic entailment relations and can lead to different query answers.

Assume the following axiom exists in the underling ontology:

$$\exists manuBy.(manu\_name = \text{“}manu_1\text{”}) \sqsubseteq Digital\_SLR.$$

This axiom states that any instance that is manufactured by the manufacturer with the name $manu_1$ is a *Digital_SLR* camera. Now suppose the two facts are also present in the underlying ontology:

$$\begin{aligned}&manuBy(a_1, m_1),\\&m_1 : manu\_name = \text{“}manu_1\text{”}.\end{aligned}$$

Under the simple entailment regime (or the RDF entailment regime), the instance a_1 is not considered to be a digital SLR camera, thus not an answer to the query in Figure 2.4. However, the instance a_1 is considered to be an instance of the class *Digital_SLR*, under the OWL 2 Direct Semantics entailment regime. In the latter entailment regime, the graph to be queried over must correspond to an OWL 2 DL ontology and the solutions to a BGP corresponds to an OWL 2 DL ontology that is entailed by the queried ontology under the OWL 2 Direct Semantics.

Dealing with blank nodes in SPARQL is subtle, because SPARQL treats blank nodes differently from the standard semantics [Mallea et al., 2011]. Formally, blanks nodes should behave as *existential* variables, which represent the existence of some anonymous resources; however, SPARQL [Prud'hommeaux and Seaborne, 2008] considers blank nodes to be constants within the scope of the graph in which they occur. Practically, a blank node in a solution is not related to the original data so that it cannot be referenced in a further query. While blank nodes are used in BGPs, they act as variables, not as references to the original blank nodes in the RDF graphs; therefore, blank nodes in queries can be replaced by variables. It is also worth noting that the combined complexity of query answering in full SPARQL is PSPACE-COMPLETE [Pérez et al., 2009], while the data complexity is LOGSPACE. SPARQL queries are very expressive, which, under bag semantics, is essentially equivalent to relational algebra [Angles and Gutierrez, 2008].

To investigate query answering over knowledge bases, a subset of FOL queries is usually studied, i.e., conjunctive queries, because FOL queries are too expressive to be managed:

query answering in FOL amounts to validity checking, which leads to undecidability. The next section discusses conjunctive queries, together with state-of-the-art techniques of efficient query answering in knowledge bases.

2.3.2 Conjunctive Queries

As discussed earlier, instance retrieval aims to find all individuals in a $\mathcal{K}$ that satisfy some query concept C. Formally, the answer to an instance retrieval query $C(x)$ for some variable x is $\{a_i \mid \mathcal{K} \models C(a_i)\}$. Also, an instance query, being the basic type of queries, is called membership atom queries [Dolby et al., 2008]. Another type of atom query, different from membership queries, asks about the relationship between individuals, i.e., $S(x, y), \{x, y\} \subseteq \mathcal{V}$, where $\mathcal{V}$ is a set of variables disjoint with the individual names in $\mathcal{K}$. Other than variables, individual names are also permitted in atom queries. In our earlier discussion, we have shown that a naïve algorithm for answering atom queries can be implemented by checking all individuals in $\mathcal{K}$, yet it is impractical for KBs with a large volume of individuals. In practice, many optimizations have to be implemented for efficient query answering.

More complex queries can be formulated based on atom queries. By convention, conjunctive queries, a fragment of SQL that is as expressive as select-project-join (SPJ) queries, are favoured as the query language for query answering in KBs.

Definition 12. ***Conjunctive Query (CQ)***. A conjunctive query $Q(\overline{x}) = \{\overline{x} \mid \bigwedge \exists \overline{y_i}.\phi_i\}$, where every ϕ_i is an atom (membership/relationship) query, or $x = y$, where $\overline{x}$ are all free (distinguished) variables occurring in ϕ_i, and $\overline{y}$ are existential (non-distinguished) variables. □

The special conjunctive query, $Q(x) = \{x \mid C(x)\}$, expresses instance queries, and $Q() = \{C(a_i)\}$ is a boolean CQ for instance checking (cf. Definition 10). Considering the inherent incompleteness of knowledge bases, given a knowledge base $\mathcal{K}$ and a query $Q(\vec{x})$, different query answers can be characterized. For instance, one can consider answers that are true in some interpretation of $\mathcal{K}$; one can also consider the answers true in *every* interpretation of $\mathcal{K}$. The former is usually called *possible answers*, and the latter *certain answers*. These query answers are formally defined below.

Definition 13. ***Possible and Certain Answers***. Given a knowledge base $\mathcal{K}$ and a query $Q(\overline{x})$ of n free variables, the possible answers and the certain answers, denoted

$\texttt{PossAns}(\mathcal{K}, Q(\overline{x}))$ and $\texttt{CertAns}(\mathcal{K}, Q(\overline{x}))$, are defined as follows, respectively:

$$\{\overline{a} \mid \mathcal{K} \cup Q(\overline{a}) \text{ is consistent or } \mathcal{K} \text{ is inconsistent}\}$$
$$\{\overline{a} \mid \mathcal{K} \models Q(\overline{a})\}$$

where $\overline{a}$ denotes a n-tuple by a substitution that maps variables in $\overline{x}$ to constants in $\mathcal{K}$. □

Possible answers can be understood as the answers that are "possibly" true, because they are true in some interpretation(s), i.e., possible answers are the union of all answers, while certain answers are the ones that are "certainly" true, i.e., certain answers are the intersection of all answers. Under CWA, say, for a complete database $\mathcal{D}$, because every n-tuple $\overline{a}$ of constants of the domain of $\mathcal{D}$ has either $\mathcal{D} \models Q(\overline{a})$ or $\mathcal{D} \models \neg Q(\overline{a})$, the possible and certain answers coincide, i.e., $\texttt{PossAns}(\mathcal{K}, Q(\overline{x})) = \texttt{CertAns}(\mathcal{K}, Q(\overline{x}))$. Because an inconsistent knowledge base can infer all facts, query answering over inconsistent KBs is meaningless. Hence, a knowledge base must be checked for consistency prior to query answering.

Data complexity of conjunctive query answering in databases is AC^0 [Abiteboul et al., 1995], while it increases to coNP-complete over expressive DL knowledge bases, such as $\mathcal{ALC}$ and its extensions [Hustadt et al., 2005; Ortiz et al., 2008]. Conjunctive query answering, in terms of data complexity, is as hard as instance retrieval in expressive DLs; however, when combined complexity is considered conjunctive query answering is harder than instance retrieval if inverse roles $\mathcal{I}$ are involved [Lutz, 2008].

Relational databases scale to large data repositories with favourable performance because of the low complexity, yet constraints are ignored during query answering. On the contrary, query answering in KBs, even ignoring constraints, still has difficulty in scaling to large data sets. Recent studies on *ontology based data access (OBDA)* [Poggi et al., 2008; Dolby et al., 2008; Calvanese et al., 2009; Heymans et al., 2008; Kontchakov et al., 2011] reconcile the complexity issue and the expressiveness in information systems. The next section then gives an overview of the state-of-the-art in OBDA.

2.3.3 Ontology Based Data Access

Ontology based data access presumes that a set of ad-hoc data sources needs to be accessed by a conceptual representation in terms of an ontology (i.e., a KB) [Poggi et al., 2008]. In such a setting, intensional knowledge (i.e., the TBox of a KB) conceptually abstracts the real data (extensional knowledge, the ABox of a KB), which is managed by technology such as the relational database. Because the real data contains values, the matching problem between the values and objects in KBs arises in OBDA. Possible solutions to the mismatch problem include approaches presented in [Poggi et al., 2008; Calvanese et al., 2009].

A corpus of literature has been devoted to scalable reasoning over large KBs, for which scalability becomes the primary objective. Some research focuses on efficiently manipulating ABox assertions through relational databases, leaving TBox constraints to DL reasoners [Zhou et al., 2006]. This approach mainly concerns novel algorithms and systems to achieve better performance on query answering in large and expressive KBs. However, the data complexity of answering conjunctive queries in expressive KBs cannot be overlooked. Another approach takes into consideration the expressiveness of KBs, and rewrites queries in a form that can be evaluated using relational technology. The following discussions focus on the second approach and considers

The *DL-Lite* Approach

The *DL-Lite* family [Artale et al., 2009; Calvanese et al., 2009] consists of a suite of light logics in the sense that, for most *DL-Lite* logics, query answering is AC^0 for data complexity and taxonomic complexity is tractable. The original motivation was to identify DLs that are capable of characterizing conceptual modelling paradigms, such as UML and ER diagrams, while enjoying tractable reasoning and low data complexity for query answering. The family of *DL-Lite* is also the basis of OWL 2 QL profile (cf. Deinition 7).

An important property of *DL-Lite* dialects is *first-order rewritability* (FO-rewritability) [Artale et al., 2009]. FO-rewritability, intuitively, is a property of a language to rewrite a user query Q into another query Q' with a given TBox $\mathcal{T}$ such that evaluating Q' over $\mathcal{A}$ returns the certain answers to Q over $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ for every $\mathcal{A}$ consistent with $\mathcal{T}$. If FO-rewritability is preserved by some logic, then it is guaranteed that query answering in that logic is in AC^0 for data complexity, permitting query answering to be delegated to the relational technology. FO-rewritability is compromised in some expressive dialects, for instance, when the *DL-Lite* logic allows individual equality to be asserted, as discussed right after Definition 7, the FO-rewritability property is forfeited in the logic.

Query answering in *DL-Lite* logics proceeds in the following way [Calvanese et al., 2009]. In the query reformulation phase, a conjunctive query Q is rewritten into a first order query Q'. The rewriting starts with atoms of each CQ, using positive inclusions (i.e., inclusions that do not have negations on the right-hand side) in the TBox as rules. The final rewritten query Q' is a union of CQs (essentially a set of CQs). In the next step, Q' is evaluated directly over the ABox data viewed as a relational database. In this phase, the TBox information is no longer used and query evaluation is the same as in relational databases. Not surprisingly, because the size of Q' does not depend on the size of the ABox data, query answering of Q' is AC^0 in data complexity, as long as FO-rewritability is preserved. In summary, the *DL-Lite* approach is to reduce query answering in KBs to query evaluation in relational databases by limiting the expressiveness of the logic to retain FO-rewritability. Again, if the logic breaks this property, such as dropping UNA and allowing for individual equality, or just allowing for role transitivity, then the *DL-Lite* approach would not work.

The Combined Approach

The *DL-Lite* approach rewrites a query with respect to a suitably tailored DL into another query that can be evaluated exclusively over the data. There are a few observations regarding the *DL-Lite* approach: the rewriting process does not modify the data; the generated union of CQs is arbitrary in size (exponential blowup of query size [Kontchakov et al., 2011]); the DLs must preserve FO-rewritability. In contrast, the *combined approach* [Kontchakov et al., 2010, 2011] pursues scalable query answering in a different manner. It is called combined because both data completion and query rewriting are used in this approach. Intuitively, data completion extends an ABox to the *canonical* model of $\mathcal{K}$, which is further encoded as a finite structure.

The combined approach first manipulates the source data stored in an ABox, and a new query is formulated based on the original query *and* the modified source data. Initially, this approach constructs the canonical model for a DL KB in *DL-Lite*$_{horn}$ [Kontchakov et al., 2011]. The canonical model is obtained by exhaustively applying the concept inclusion constraints in the TBox to the ABox data, *irrespective* of any particular query. Essentially, marked nulls are introduced in the canonical model, which is preprocessed and stored in the databases. The canonical model may be infinite; however, it can be encoded in a finite FOL structure called the generating model. Roughly, the canonical model can be considered as the unraveling model of the generating model. The generating model may produce bogus answers; hence, to purge these answers, Q over the original KB is reformulated into another query Q' polynomial in the size of Q. The answers to Q' over the generating model are exactly the certain answers to Q over the initial $\mathcal{K}$.

Two advantages over the *DL-Lite* approach can be observed in the combined approach. First, the size of the rewritten query is bounded and much smaller than that of the reformulated queries in the *DL-Lite* approach, which allows for efficient execution by relational DBMSs. In addition, the logic used in describing a TBox does not have to retain FO-rewritability to ensure data complexity in AC^0 for query answering. Kontchakov et al.

[2011] suggest that logics with query answering in PTIME-COMPLETE can take advantage of this combined approach.

The Datalog Approach

Datalog is a database query language that captures recursions based on logic programming paradigm [Abiteboul et al., 1995]. In Datalog programs, both assertions (a.k.a. *facts*) and rules are represented as (universally quantified) Horn clauses (a disjunction of literals of which at most one is positive); for example, a rule about reachability is represented as follows:

$$reachable(z, x) :- reachable(z, y), reachable(y, x).$$

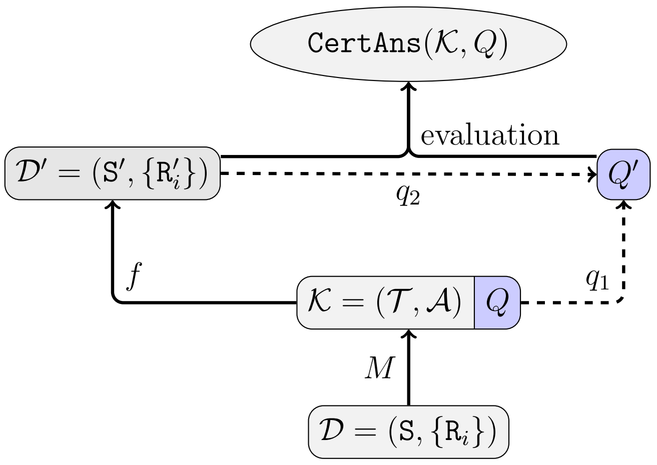
The semantics of a Datalog program is determined by a *minimal* model. The idea for choosing a minimal model is closely related to CWA: the intended model consists of the facts we know must be true in all models and should not contain more ground facts than necessary for satisfying the program. It is easy to see that negations and disjunctions are difficult to deal with under this semantics. Indeed, Datalog programs cannot express negations, i.e., the *difference* operator in relational algebra, because of the Horn restrictions for datalog programs. However, Datalog supports recursive queries, which cannot be handled by relational algebra.

Recent research has proposed a family of Datalog variants, called Datalog$^\pm$, that adds the capabilities for existential restrictions, the equality predicate, and $\perp$ (i.e., the constant false) [Cal et al., 2010]. This Datalog$^\pm$ family is studied for tractable query answering w.r.t. data complexity. In particular, the guarded Datalog$^\pm$, in which there is an atom in the rule body that contains all universally quantified variables, ensures decidability and is tractable in the data complexity. Another even more restricted, guarded variant is the linear Datalog$^\pm$, in which query answering has the FO-rewritability: only a singleton body atom is allowed. More variants, including guarded and sticky variants, are discussed in [Cal et al., 2010].

In the context of ontological query answering, researchers have considered query rewriting on *boolean* conjunctive queries [Gottlob et al., 2011] in Datalog programs. The rewriting of a boolean CQ is computed by exhaustively expanding the query body atoms using the applicable constraints. Each application of a constraint results in a new query in a backward-chaining fashion, with the use of unification of atoms; this process is repeated until no new queries are generated. For every newly generated query, it is checked in the factorization step to find if its size can be reduced through unification of atoms. The set of all generated queries is the perfect rewriting of the original query w.r.t. the ontology represented by the Datalog program. Note that this approach yields exponentially large CQs. In [Gottlob and Schwentick, 2012], it is shown that, however, a *polynomially* sized non-recursive Datalog can be obtained, which can in turn be efficiently translated into SQL statements to be evaluated directly by relational DBMSs.

Comparing Current OBDA Approaches

The previously discussed approaches are the representative OBDA methods, sharing a few commonalities and differing in several other aspects. Specifically, the reformulation (and the query answering) process is abstracted in Figure 2.5. A knowledge base consists of a TBox $\mathcal{T}$ (or Datalog constraints) that provides vocabulary for describing the data and an ABox $\mathcal{A}$ that is assumed to be stored in relational DBMSs, or, w.l.o.g, is obtained by some mapping M that ports data from other data sources, such as a legacy database $\mathcal{D}$.



$\texttt{CertAns}(\mathcal{K}, Q)$
evaluation
$\mathcal{D}' = (\texttt{S}', \{\texttt{R}'_i\})$
Q'
q_2
f
q_1
$\mathcal{K} = (\mathcal{T}, \mathcal{A})$
Q
M
$\mathcal{D} = (\texttt{S}, \{\texttt{R}_i\})$

Figure 2.5: Approaches to OBDA. $\texttt{S}$, $\texttt{S}'$ ($\texttt{R}_i$, $\texttt{R}'_i$) denote the schemata (relation names) in relational databases $\mathcal{D}$ and $\mathcal{D}'$ resp., f some function for KB manipulation, q_i some function for query reformulation, M some mapping, and Q, Q' queries.

These approaches reformulate the original CQ Q into another (CQ or non-CQ) query Q'. The fundamental idea for all the aforementioned approaches is the reduction of query answering over knowledge bases to query evaluation in relational DBMSs: the *certain answers* to Q over $\mathcal{K}$ are exactly the answers to Q' over $\mathcal{D}'$. Table 2.1 describes the differences between the OBDA approaches.

Note that the Datalog approach are similar to the *DL-Lite* approach regarding the

Features	*The DL-Lite Approach**	*The Combined Approach*
logic	data complexity AC^0	data complexity PTIME-COMPLETE
$\mathcal{T}$	used to rewrite Q	*not* used to rewrite Q
f	the identity function	uses $\mathcal{T}$ to modify $\mathcal{A}$ by introducing marked nulls
q_i	uses certain inclusions from $\mathcal{T}$ to rewrite Q, i.e., q_1	uses modified $\mathcal{A}$ to rewrite Q, i.e., q_2
$\mathcal{D}'$	$\mathcal{D}' = \mathcal{A}$	$\mathcal{D}'$ is the generating model polynomial in size of $\mathcal{K}$
Q'	union of CQs, may be exponential in size of Q	not CQs (e.g., with disjunctions and negations), polynomial in size of Q

Table 2.1: Comparing characteristics of the approaches to OBDA.

features in Table 2.1, except that in the Datalog approach the input query is a boolean CQ and the languages considered are specific variants in the Datalog$^{\pm}$ family.

An important observation for all the aforementioned OBDA approaches is that the dialects used in all these approaches cannot support any non-Horn extensions, for instance, that include disjunctions. For large knowledge bases with non-Horn features, new approaches to query answering need to be developed.

2.4 Query Optimization

Query processing in relational database systems is sophisticated, and, in general, consists of two phases: compilation and execution. The first phase is analogous to compiling program source code into object code and the second phase is analogous to executing the object code generated in the first phase. Specifically for database systems, query compilation translates query specification into executable code that involves predefined operations. The compilation phase can be further divided into several steps: lexical and syntactic analyses that parse the user queries in some representation forms such as parse trees, query rewriting that converts parse trees into an initial algebraic expression and likely other equivalent expressions, and code generation that turns the previous algebraic expressions into executable *query plans*. The algebraic expressions generated in query compilation are called *logical query plans*, while the corresponding code generated in the latter step forms *physical query plans* and can be executed directly by database systems. The process that translates a parse tree into final physical plans is called *query optimization*, and the program that fulfills these tasks in database systems is a *query optimizer* [Markl, 2009; Molina et al., 1999].

Query compilation, particularly query optimization, has been a core research problem since the inception of the relational model. The goal of a query optimizer is to derive "good" physical query plans for execution based on certain performance measures. First, a query optimizer needs to define a space of candidate query plans; subsequently, the optimizer requires a cost estimation technique such that relevant measures, such as IO cost, memory usage, query response time and so on, are reflected in the cost of executing each plan. Finally, the query optimizer must use an efficient algorithm to enumerate plans in the search space based on the estimated cost. It is clear that a query can have many equivalent logical plans (e.g., by commuting join orders), and each logical plan can be implemented by different physical operators, resulting in multiple physical plans; for instance, a variety of join algorithms, such as nested loops join and sort-merge join, can be adopted to implement a join operation. Relevant techniques can be found in the literature, e.g., [Chaudhuri, 1998].

Selinger et al. [1979] proposed the classical query optimizer in System R, which has been used successfully in commercial database systems including IBM DB2 and Oracle. The query optimizer of System R first determines the cheapest base table access paths, then it tries all possible ways of joining the base access paths to get two-table plans. The query optimizer repeats for all other table joins and associate each query plan a cost expressed as a combination of measures like intermediate result sizes, among others. To prune the search space of plans, dynamic programming is employed to retain plans with the lowest cost. An additional feature of System R query optimization is to keep also interesting plans that may be exploited later to achieve a final plan with possibly lower cost; for instance, a plan that has a particular sort for an intermediate result set. In contrast to the bottom-up approach in System R, Graefe and McKenna [1993] proposed an alternative top-down approach to query optimization in the Volcano system, which has been implemented in DBMSs such as Microsoft's SQL Server. Volcano uses the goal-directed dynamic programming search to find query plans; particularly, it does not generate all promising/interesting plans as in System R, instead, it only derives plans that actually participate in larger plans. In short, the top-down approach maintains only feasible execution plans, so, a top-down optimizer may stop at any time while obtaining an executable plan; in contrast, a bottom-up optimizer may not produce any feasible plans even after exceeding resource limits.

For both top-down and bottom-up approaches to query optimization, the repertoire of rewriting rules is essential for generating and expanding plans. Query rewriting takes place in many forms; for instance, it can rewrite predicates to enable index-based access, remove redundant predicates and joins, and un-nest subqueries for simplification. More interestingly, if there are materialized views (essentially named queries with results) in the databases, query rewriting can exploit views to answer queries. *View-based query rewriting* has been a central problem for information integration from heterogeneous data sources. More generally, query folding [Qian, 1996], i.e., answering queries using a given set of resources, is a generalization of view-based query rewriting.

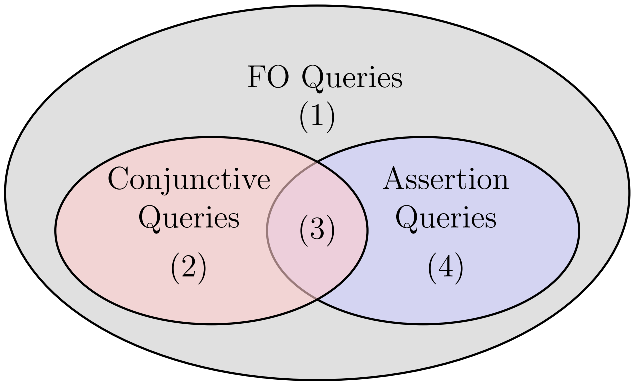


Figure 2.6: View-based query rewriting.

View-based query rewriting has been extensively studied in the literature, e.g., [Halevy, 2001]. Query containment, which decides if the results of one query is a subset of the results of another, is the fundamental task for view-based query rewriting. Figure 2.6 shows the different types of queries used to define views in view-based query rewriting. In the relational setting, research has focused on views defined by conjunctive queries for view-based query rewriting, while, in this work, Chapter 5 shows how cached assertion queries can be leveraged for view-based query rewriting. Conjunctive queries in the area (2) in Figure 2.6 denote queries that have free variables, which are not expressible in assertion queries; while assertion queries in (4) are those that support left-outer join operations. The intersection area (3) then denotes queries that are one-variable conjunctive queries, for example, instance queries. In general, arbitrary first order queries can be used to define views, i.e., (1). Because Chapter 5 considers only views that are defined by assertion queries, this work is incomparable to view-based query rewriting studied in relational databases.

2.4.1 Query Optimization in DL Knowledge Bases

Traditionally, an instance checking task of the form $\mathcal{K} \models a : C$ is reduced to consistency checking of $\mathcal{K} \cup \{a : \neg C\}$. Consistency is verified by computing a *pre-completion* that contains no contradictions (a.k.a. clashes) via an exhaustive application of non-generating expansion rules. Recall that a completion graph is obtained by applying all expansion rules according to the tableau algorithm, a pre-completion is simply a unfinished completion graph. A pre-completion captures *some* of the relevant information regarding a w.r.t. $\mathcal{K}$. For DLs without disjunctions, only one pre-completion will be obtained; however, a pre-completion has to be guessed non-deterministically for DLs that involve disjunctions. A similar idea, called the *model merging* technique, was presented in [Horrocks, 1997] for more expressive DLs. The model merging strategy uses pre-completion obtained by conjunction and disjunction rules, and has been further refined to *pseudo model merging* [Haarslev and Möller, 2008], which appeared to be indispensable for answering queries over large ABoxes.

Notably, the pseudo model merging technique separates consistency checking of $\mathcal{A}$ from instance checking by avoiding exploring other instances occurring in $\mathcal{A}$. Specifically, given a *consistent* $\mathcal{A}$, a pseudo model of an instance a captures the deterministic information relevant to a in one model of $\mathcal{A}$ computed by the tableaux algorithm, which exhibits the interaction between other instances and a. Observing that an instance usually belongs to a small number of concepts, pseudo models are often used to exclude obvious non-instances of a given concept. The experimental evaluation of [Haarslev and Möller, 2008] demonstrated the usefulness of pseudo model merging; however, it is a sound but incomplete optimization.

Although it is always possible to evaluate an instance query $C(x)$ by performing a sequence of instance checks $\mathcal{K} \models a : C$ for each individual a occurring in $\mathcal{K}$, reasoning engines usually try to reduce the number of such checks by using precomputed results or by "bulk processing" of a range of instance checks. An example of the latter is so-called *binary retrieval* [Haarslev and Möller, 2008], which is used to determine non-answers via a single (possibly large) satisfiability check. Another approach to avoiding checking individuals sequentially is through summarization and refinement [Dolby et al., 2007], in which query evaluation is performed over a summary of the original data (ABox), and iterative refinement based on inconsistency justification is used to purge spurious answers. Observe that the aforementioned optimizations concern how to reduce the number of instance checking tasks, not how to improve the instance checking problem itself. An approach to optimizing instance checking was introduced in [Wandelt and Möller, 2012]. In this case, an ABox is partitioned into small islands such that an instance checking problem that involves only atomic concepts is routed to the island "owned" by an individual.

There has also been a body of research on query optimization in RDF/RDFS datasets.

For instance, Neumann and Weikum [2010] presents a RDF data management system, RDF3X, that leverages conventional relational-style query optimization for query answering. In fact, commercial database engines, such as DB2 [Bornea et al., 2013] and Oracle [Kolovski et al., 2010], also support RDF data management with native relational query optimization. When ontologies, rather than RDF/RDFS datasets, are involved in query answering, sophisticated inferences are necessary. In these cases, more straightforward optimizations are in spirit to indexing in relational technology for scalability; prominent examples include [Haarslev and Möller, 2008; Kollia and Glimm, 2012]. Typically, indexes are created for instances belonging to certain concepts such that answering queries that involve these concepts is efficient. In general, an instance is indexed by its most specific concepts. The concepts used in indices can range from only atomic concepts to arbitrary concepts, depending on the application. For binary relations, i.e., roles, indexing can support queries on role predecessors, successors or both [Kollia and Glimm, 2012]. In fact, pseudo model merging can be viewed as a way of manipulating a set of aforementioned indices to efficiently compute instance queries.

Chapter 3

User Queries and Query Plans

Recall from the introductory comments in Chapter 1 that object queries qualify a subset of objects of interest to users. In knowledge bases object queries are typified by instance retrieval (see Definition 10), which returns objects with object identifiers. These query answers are insufficient because object identifiers are usually not informative, as they do not capture the information content about objects, while users are often interested in knowing particular properties of the qualifying objects, such as the price of a product.

A more general kind of object queries can retrieve qualifying objects and depict objects with additional information *about* these objects. Such a paradigm addresses two drawbacks of instance retrieval, i.e., the inability to express the *explicit object characteristics* of interest and the inability to retrieve objects that *implicitly* satisfy query predicates by means of schema-based reasoning. This chapter proposes a new query paradigm, *assertion retrieval*, as a more preferred for object retrieval. The subsequent sections introduce the syntax and semantics of user queries, i.e., assertion queries, in Section 3.1, details about the projection operator that are indispensable for formulating a user query in Section 3.2, the comparison between assertion queries and SPARQL queries and conjunctive queries in Section 3.3, the procedures for computing projections with the correctness proof in Section 3.4, and the algebraic operators that can be used to express query plans in Section 3.5.

3.1 Assertion Queries

An assertion query (abbreviated as AQ) is now formulated by a pair (C, Pd) in which C is a query concept in some DL $\mathfrak{L}$ serving the same role as in instance retrieval, and in which

Pd, a *projection description*, defines a subset $\mathfrak{L}_{Pd}$ of the $\mathfrak{L}$-concepts. The query answers to an assertion query is of the form $a : C_a$, where a is the identifier and the concept C_a in $\mathfrak{L}$ is "the most informative" concept about a that can be expressed in a subset of $\mathfrak{L}$-concepts defined by the projection description.

For example, the query ($Student$, age?) extends the retrieval of student instances by adding the projection description age? requesting more information about the age of all qualifying students. The answers to such queries are assertions of the form $a : C_a$ such that $\mathcal{K} \models a : (C \sqcap C_a)$, where C_a is the *most specific concept* in $\mathfrak{L}_{Pd}$ for which the above logical consequence holds. For example, an answer could be, ignoring the intricacies of computing $\mathfrak{L}_{Pd}$ for now, a_1: $(age = 26)$ for the student identified by a_1. A projection description generalizes the effect of the relational *projection operation* by providing a more general way of controlling both the information content and the format of query results [Pound et al., 2009, 2010].

There are two compelling reasons for considering assertion retrieval over simple instance retrieval. The first is a practical reason that relates to usability: since it becomes possible to include relevant facts about objects in additional to their identifiers, the results of querying can be more informative and relevant to a user browsing a knowledge base. The second is a more technical reason that relates to performance: by caching the computation of earlier assertion retrieval queries, it becomes possible to explore view-based query rewriting (see Section 5.1) in the context of object queries over web data.

3.2 The Projection Description

Recall that a user query (C, Pd) consists of a query concept C paired with a projection description Pd. In particular, an instance query C over $\mathcal{K}$ can be formulated as query $(C, \top?)$ (effectively retrieving no further information about qualifying individuals in $\mathcal{K}$). The syntax for Pd and the sublanguage of concepts in some DL dialect $\mathfrak{L}$ that are induced by a Pd are defined as follows. Note that given a finite set S of $\mathfrak{L}$ concepts, when $S = \{D_1, ..., D_n\}$, $\sqcap S$ is written to denote $\top$ if S is empty and the concept $D_1 \sqcap \cdots \sqcap D_n$ otherwise.

Definition 14. ***Projection Description and Induced Concepts***. Let $f \in$ NF, $R \in$ NR and $C \in$ NC be a concrete feature, role and concept, respectively. A projection description Pd is defined by the grammar:

$$Pd ::= C? \mid f? \mid Pd_1 \sqcap Pd_2 \mid \exists R.Pd$$

We define the sets $\mathcal{L}_{Pd}$ and $\mathcal{L}_{Pd}^{\text{TUP}}$, the $\mathcal{L}$ *concepts* generated by Pd and $\mathcal{L}$ *tuple concepts* generated by Pd, respectively, as follows:

$$\mathcal{L}_{Pd} \;=\; \{\sqcap S \mid S \subseteq_{\text{fin}} \mathcal{L}_{Pd}^{\text{TUP}}\}, \text{ where } \subseteq_{\text{fin}} \text{ denotes finite subsets, and}$$

$$\mathcal{L}_{Pd}^{\text{TUP}} \;=\; \begin{cases} \{C, \top\} & \text{if } Pd = C?; \\ \{f = k \mid k \in \mathbb{D}\} \cup \{\top\} & \text{if } Pd = f?; \\ \{C_1 \sqcap C_2 \mid C_1 \in \mathcal{L}_{Pd_1}^{\text{TUP}}, C_2 \in \mathcal{L}_{Pd_2}^{\text{TUP}}\} & \text{if } Pd = Pd_1 \sqcap Pd_2; \text{ and} \\ \{\exists R.C \mid C \in \mathcal{L}_{Pd_1}\} & \text{if } Pd = \exists R.Pd_1. \end{cases}$$

□

Although no restrictions are imposed on $\mathfrak{L}$, i.e., the DL dialect from which the $\mathcal{L}_{Pd}$ and $\mathcal{L}_{Pd}^{\text{TUP}}$ concepts are induced, the syntax of Pd requires that $\mathfrak{L}$ be at least as expressive as $\mathcal{EL}(\mathbb{D})$. The two induced sets of concepts $\mathcal{L}_{Pd}$ and $\mathcal{L}_{Pd}^{\text{TUP}}$ in Definition 14 are correlated in that the latter is a proper subset of the former (modulo semantic equivalence). In addition, $\mathcal{L}_{Pd}$ has the following property:

Lemma 3.2.1. *For any given Pd, any pair of concepts $\{D_1, D_2\} \subseteq \mathcal{L}_{Pd}$ and $\not\models D_1 \sqcap D_2 \sqsubseteq \bot$ (i.e., $D_1 \sqcap D_2$ is a satisfiable concept), there is $D_3 \in \mathcal{L}_{Pd}$ such that $\models D_3 \equiv D_1 \sqcap D_2$.*

Proof. We prove this claim by structural induction on Pd.

- When $Pd = C?$, $\mathcal{L}_{Pd} = \{C, \top, C \sqcap \top\}$, so it is easy to see the claim holds, e.g., by allowing D_3 to be either C or $\top$.
- When $Pd = f?$, because $\mathcal{K} \models D_1 \sqcap D_2 \not\sqsubseteq \bot$, it is impossible for both $f = k_1$ and $f = k_2$ to occur in $D_1 \sqcap D_2$, where $k_1 \neq k_2$ (otherwise $D_1 \sqcap D_2$ is unsatisfiable). So, if $f = k$ occurs in $D_1 \sqcap D_2$, then $D_3 = (f = k)$, otherwise $D_3 = \top$.
- When $Pd = Pd_1 \sqcap Pd_2$, by induction hypotheses, there is $D_3^i \in \mathcal{L}_{Pd_i}$ such that $\models D_3^i \equiv D_1 \sqcap D_2$ for any consistent pair $\{D_1, D_2\} \subseteq \mathcal{L}_{Pd_i}$, $i \in \{1, 2\}$. By Definition 14, let D_3 be $D_3^1 \sqcap D_3^2$, then $D_3 \in \mathcal{L}_{Pd}$ and $\models D_3 \equiv D_1 \sqcap D_2$ for any consistent pair $\{D_1, D_2\} \subseteq \mathcal{L}_{Pd}$.
- When $Pd = \exists R.Pd_1$, any concept $D \in \mathcal{L}_{Pd}$ can be represented as $\bigsqcap \exists R.C$ (or $\top$), where $C \in \mathcal{L}_{Pd_1}$. For any consistent pair $\{D_1, D_2\} \subseteq \mathcal{L}_{Pd}$, clearly, $D_1 \sqcap D_2$ can also be represented as $\bigsqcap \exists R.C$, where each $\exists R.C$ occurs either in D_1 or D_2 and $C \in \mathcal{L}_{Pd_1}$. Hence, $D_3 \equiv D_1 \sqcap D_2 \in \mathcal{L}_{Pd}$.

□

Intuitively, Lemmas 3.2.1 states that satisfiable concepts in the induced set of concepts $\mathcal{L}_{Pd}$ are closed under conjunction, up to semantic equivalence. The closure property will be used later to facilitate efficient computation of $\mathcal{L}_{Pd}$. There is another useful property for computing $\mathcal{L}_{Pd}$, when $Pd = Pd_1 \sqcap Pd_2$. First, the relationship of semantic subset is defined to consider sets of concepts by semantic equivalence.

Definition 15. ***Semantic Subset***. For any two sets of concepts S_1 and S_2, S_1 is a *semantic subset* of S_2, denoted $S_1 \hookrightarrow S_2$, if for every *satisfiable* concept $C_1 \in S_1$ there is $C_2 \in S_2$ such that $\models C_1 \equiv C_2$. □

It is easy to see that $\hookrightarrow$ is transitive. Given Definition 15, it is possible to consider only satisfiable concepts in a set of concepts and establish connections between such sets.

Lemma 3.2.2. *For $Pd = Pd_1 \sqcap Pd_2$, let S denote the set of concepts $\{C_1 \sqcap C_2 \mid C_i \in \mathcal{L}_{Pd_i}, i \in \{1,2\}\}$, then $\mathcal{L}_{Pd} \hookrightarrow S$ and $S \hookrightarrow \mathcal{L}_{Pd}$.*

Proof. The direction $S \hookrightarrow \mathcal{L}_{Pd}$ follows from Lemmas 3.2.1. Because $\mathcal{L}_{Pd_i}, i \in \{1,2\}$ are subsets of $\mathcal{L}_{Pd}$ up to semantic equivalence, since $\top \in \mathcal{L}_{Pd'}$ for any Pd'.

For the other direction, let $C \in \mathcal{L}_{Pd}$ and C is satisfiable. By Definition 14, $C \equiv \bigsqcap D_j, D_j \in \mathcal{L}^{\mathrm{TUP}}_{Pd}$. Again, the definition of $\mathcal{L}^{\mathrm{TUP}}_{Pd}$ implies $C \equiv \bigsqcap D^1_j \sqcap \bigsqcap D^2_j$, where $D^i_j \in \mathcal{L}^{\mathrm{TUP}}_{Pd_i}, i \in \{1,2\}$. Since $\mathcal{L}^{\mathrm{TUP}}_{Pd_i} \subseteq \mathcal{L}_{Pd}, i \in \{1,2\}$, together with Lemmas 3.2.1, it follows that $C \equiv C_1 \sqcap C_2$, where $C_i \in \mathcal{L}_{Pd_i}, i \in \{1,2\}$. This shows $C \in S$. □

For a given Pd, any concept occurring in $\mathcal{L}_{Pd}$ satisfies a syntactic format *conforming* to Pd independently of any knowledge base $\mathcal{K}$. However, not all syntactically "correct" concepts, i.e., all concepts in $\mathcal{L}_{Pd}$, should qualify as answers. Concepts that merit consideration are those satisfying certain properties, and we define such promising concepts to be the *most informative* concepts in a set with respect to a given $\mathcal{K}$. Given a knowledge base $\mathcal{K}$ and set of concepts S, the most informative concepts in S with respect to $\mathcal{K}$ are thus defined below:

Definition 16. ***Most Informative Concepts***. Let S and $\mathcal{K}$ be a set of concepts and knowledge base, respectively. We write $\lfloor S \rfloor_{\mathcal{K}}$ to denote $\{C \in S \mid \neg\exists D \in S : (\mathcal{K} \models D \sqsubseteq C, \mathcal{K} \not\models C \sqsubseteq D)\}$. □

The reduction $\lfloor S \rfloor_{\mathcal{K}}$ of S, the set of syntactically desirable concepts, retains representative concepts in this set via the use of semantic equivalence w.r.t. $\mathcal{K}$. Note that the reduction can retain more than one concept because of equivalence. A further reduction of these concept, i.e., $\lfloor \lfloor S \rfloor_{\mathcal{K}} \rfloor_{\emptyset}$, can then retain the most informative concepts w.r.t. $\emptyset$.

Lemma 3.2.3. *Let $\mathcal{K}$ be an $\mathfrak{L}$ knowledge base for some DL dialect $\mathfrak{L}$ that is at least as expressive as $\mathcal{EL}(\mathbb{D})$, Pd a projection description and C a concept. Then the following hold for the set S of concepts defined by $\{D \in \mathcal{L}_{Pd} \mid \mathcal{K} \models C \sqsubseteq D\}$:*

1. *$\lfloor S \rfloor_{\mathcal{K}}$ is non-empty;*
2. *$\mathcal{K} \models C_1 \equiv C_2$, for any $\{C_1, C_2\} \subseteq \lfloor S \rfloor_{\mathcal{K}}$; and*
3. *$\lfloor \lfloor S \rfloor_{\mathcal{K}} \rfloor_{\emptyset}$ is non-empty,*

Proof. Lemma 3.2.3 (1). Observe that the definition of S ensures that S is non-empty because $\top \in S$ by the definition of $\mathcal{L}_{Pd}$. Assume that $\lfloor S \rfloor_{\mathcal{K}}$ is empty. By Definition 16, for every concept $C \in S$ there exists $D \in S$ such that $\mathcal{K} \models D \sqsubseteq C$ and $\mathcal{K} \not\models C \sqsubseteq D$. Given the finiteness of S, we assume there are n concepts $D_1, \cdots, D_n$ in S. For D_1, there is another concept, say, D_2, in S such that $\mathcal{K} \models D_2 \sqsubseteq D_1$ and $\mathcal{K} \not\models D_1 \sqsubseteq D_2$. By doing so for every concept in S, we can order all the concepts in S such that $\mathcal{K} \models D_{i+1} \sqsubseteq D_i$ and $\mathcal{K} \not\models D_i \sqsubseteq D_{i+1}$ for $1 \leq i \leq n-1$. Finally, for D_n we must also find another concept D_m ($1 \leq m < n$) such that $\mathcal{K} \models D_m \sqsubseteq D_n$ and $\mathcal{K} \not\models D_n \sqsubseteq D_m$; however, it can be inferred that $\mathcal{K} \models D_n \sqsubseteq D_m$ because $m < n$, which contradicts the assumption. Therefore, $\lfloor S \rfloor_{\mathcal{K}}$ must be non-empty.

Lemma 3.2.3 (2). We prove this conclusion by structural induction on Pd. Let $S = \{D \in \mathcal{L}_{Pd} \mid \mathcal{K} \models C' \sqsubseteq D\}$ for the user query (C', Pd).

- Case "C?". $\mathcal{L}_{Pd} = \{C, \top, C \sqcap \top\}$. Assume $\mathcal{K} \models C' \sqsubseteq C$ (otherwise $\lfloor S \rfloor_{\mathcal{K}} = \{\top\}$), then $\lfloor S \rfloor_{\mathcal{K}} = \{C, C \sqcap \top\}$. Clearly, $\mathcal{K} \models C \equiv (C \sqcap \top)$.
- Case "f?". Similar to the above case.
- Case "$Pd_1 \sqcap Pd_2$". By the induction hypothesis the second conclusion holds in both cases of Pd_1 and Pd_2. That is, $\lfloor S_1 \rfloor_{\mathcal{K}}$ and $\lfloor S_2 \rfloor_{\mathcal{K}}$ for Pd_1 and Pd_2 resp. contain only equivalent concepts w.r.t. $\mathcal{K}$. It can be shown that any concept C in $\lfloor S \rfloor_{\mathcal{K}}$ are composed of minimal subconcepts from $\lfloor S_1 \rfloor_{\mathcal{K}}$ and $\lfloor S_2 \rfloor_{\mathcal{K}}$, otherwise C cannot be a minimal concept in $\lfloor S \rfloor_{\mathcal{K}}$. Suppose w.l.o.g. that for any two concepts ($C_1 = A_{S_1} \sqcap A_{S_2}$) $\in \lfloor S \rfloor_{\mathcal{K}}$ and ($C_2 = B_{S_1} \sqcap B_{S_2}$) $\in \lfloor S \rfloor_{\mathcal{K}}$, with $\{A_{S_1}, B_{S_1}\} \subseteq \lfloor S_1 \rfloor_{\mathcal{K}}$ and $\{A_{S_2}, B_{S_2}\} \subseteq \lfloor S_2 \rfloor_{\mathcal{K}}$. Note that C_1 (and C_2) can be a conjunction of more than two concepts by the definition of $\mathcal{L}_{Pd}Pd$, then in such cases we always rearrange

the conjunctions to have two equivalent concepts renamed as A_{S_1} and A_{S_2}, where A_{S_1} (resp. A_{S_2}) only includes concepts in $\lfloor S_1 \rfloor_\mathcal{K}$ (resp. $\lfloor S_2 \rfloor_\mathcal{K}$). By the induction hypothesis, $\mathcal{K} \models A_{S_1} \equiv B_{S_1}$ and $\mathcal{K} \models A_{S_2} \equiv B_{S_2}$, which implies $\mathcal{K} \models C_1 \equiv C_2$.

- Case "$\exists R.Pd_1$". By the induction hypothesis the second conclusion holds in the case of Pd_1. The conclusion that any concept C in $\lfloor S \rfloor_\mathcal{K}$ are composed of minimal subconcepts from $\lfloor S_1 \rfloor_\mathcal{K}$ still holds because $A \sqsubseteq B$ implies $\exists R.A \sqsubseteq \exists R.B$. Consequently, suppose w.l.o.g. that for any two concepts $(C_1 = \exists R.A_{S_1}) \in \lfloor S \rfloor_\mathcal{K}$ and $(C_2 = \exists R.B_{S_1}) \in \lfloor S \rfloor_\mathcal{K}$, it follows that $\{A_{S_1}, B_{S_1}\} \subseteq \lfloor S_1 \rfloor_\mathcal{K}$. By the induction hypothesis, $\mathcal{K} \models A_{S_1} \equiv B_{S_1}$, which implies $\mathcal{K} \models C_1 \equiv C_2$.

Lemma 3.2.3 (3). Analogous to the proof of Lemma 3.2.3 (1), it is easy to show that $\lfloor \lfloor S \rfloor_\mathcal{K} \rfloor_\emptyset$ is also non-empty. □

Lemma 3.2.3 (1) and (2) ensure that at least one least subsuming concept of C exists in $\mathcal{L}_{Pd}$ and, when there is more than one, that any pair are semantically equivalent with respect to a given $\mathcal{K}$. Note that such $\mathcal{L}$ restriction of some DL dialect $\mathfrak{L}$ is essential to ensure Lemma 3.2.3 (1); for example, a more general $\mathcal{L}$ restriction that excludes concept negation from $\mathcal{ALCI}(\mathbb{D})$ may not have this property [Baader et al., 2007]. In addition, although $\mathcal{L}_{Pd}$ is infinite in general, for any fixed and finite terminology $\mathcal{K}$ and query concept C, the language $\mathcal{L}_{Pd}$ restricted to the symbols used in $\mathcal{K}$ and C is necessarily finite. We elaborate on issue of finiteness when attempting to use $\mathcal{L}_{Pd}$ in finding query plans (see Section 5.2). Lemma 3.2.3 (3) ensures that, among the least subsuming concepts of C in $\mathcal{L}_{Pd}$ with respect to $\mathcal{K}$, there is at least one least subsuming concept that is *the most informative* when no knowledge of $\mathcal{K}$ is presumed.

Since in general only one concept in the set $\lfloor \lfloor S \rfloor_\mathcal{K} \rfloor_\emptyset$ is needed, $\llfloor S \rrfloor_\mathcal{K}$ is written to denote the *minimum* concept from the set of concepts $\lfloor \lfloor S \rfloor_\mathcal{K} \rfloor_\emptyset$ according to an *arbitrary total ordering* of all concepts in the DL fragment $\mathfrak{L}$.

Example 2. Let $\mathcal{K} = \{\mathit{Sophomore} \sqsubseteq \mathit{year} = 2\}$ and $Pd = (\mathit{Sophomore}? \sqcap \mathit{year}?)$, and let $S = \{C \in \mathcal{L}_{Pd} \mid \mathcal{K} \models \mathit{Sophomore} \sqsubseteq C\}$. Then

1. $\lfloor S \rfloor_\mathcal{K} = \{\mathit{Sophomore} \sqcap (\mathit{year} = 2), \mathit{Sophomore} \sqcap \top\}$,
2. $\lfloor \lfloor S \rfloor_\mathcal{K} \rfloor_\emptyset = \{\mathit{Sophomore} \sqcap (\mathit{year} = 2)\}$ and
3. $\llfloor S \rrfloor_\mathcal{K} = \mathit{Sophomore} \sqcap (\mathit{year} = 2)$.

□

The induced concepts of Pd and the most informative concept can be used to define the semantics of an assertion query. The syntax of a user query, together with its semantics, now follow.

Definition 17. ***Query Syntax and Semantics***. Let $\mathcal{K}$ be an $\mathfrak{L}$ knowledge base for some DL fragment $\mathfrak{L}$ that is at least as expressive as $\mathcal{EL}(\mathbb{D})$, a user query Q over $\mathcal{K}$ is a pair (C, Pd), where C is a $\mathfrak{L}$-concept and Pd is a projection description. Let $\textsc{Eval}(Q, \mathcal{K})$ denote the set of concept assertions computed by Q, then $\textsc{Eval}(Q, \mathcal{K}) = \{a : \llbracket\{D \mid D \in \mathcal{L}_{Pd}, \mathcal{K} \models a : D\}\rrbracket_{\mathcal{K}} \mid \mathcal{K} \models a : C, a \text{ occurs in } \mathcal{K}\}$. □

Definition 17 shows that a user query in assertion retrieval is reduced to *instance checking* in any DL fragment $\mathfrak{L}$ that has the expressivity of $\mathcal{EL}(\mathbb{D})$ and above. Therefore, any existing DL reasoner that supports $\mathcal{EL}(\mathbb{D})$ and more expressive languages can be used to answer such a user query. Conversely, instance retrieval reduces to a special case of assertion retrieval.

3.3 Comparing Query Languages

Assertion queries extends the well-known instance queries in DL knowledge bases, however, the relation between AQ and other popular query languages, such as SPARQL and conjunctive queries, is nebulous to this end. This section illustrates the specific properties that are unavailable in SPARQL or CQ yet are specific to AQ.

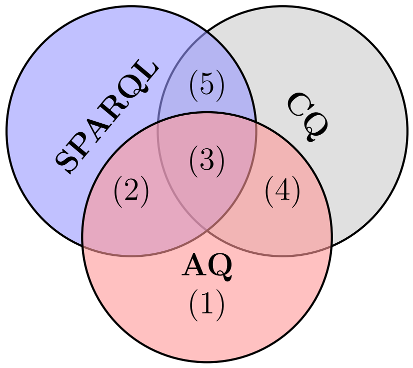


Figure 3.1: The relationship between the semantics of SPARQL queries, conjunctive queries (CQ), and assertion queries (AQ).

Figure 3.1 depicts the differences existing in the semantics of the three query languages. In particular, the five identified areas are elaborated in the subsequent discussions. Example 3 first illustrates the computing of nested relations according to the semantics of each query language.

Example 3. Consider the following knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, where $\mathcal{T}$ and $\mathcal{A}$ are defined as follows:

$$\begin{aligned}
\mathcal{T} = \{\ & \mathit{AZProduct} \sqsubseteq \exists \mathit{soldBy}.(\mathit{name} = \text{“}AZ\text{”})\ \} \\
\mathcal{A} = \{\ & p_1 : \mathit{price} = 50 \quad p_2 : \mathit{price} = 60 \quad p_3 : \mathit{price} = 70 \quad p_4 : \mathit{price} = 80 \sqcap \mathit{AZProduct} \\
& r_1 : \mathit{Reseller} \sqcap \mathit{name} = \text{“}FS\text{”} \quad r_2 : \mathit{Reseller} \sqcap \mathit{name} = \text{“}BB\text{”} \quad r_3 : \mathit{Reseller} \\
& \mathit{soldBy}(p_1, r_1) \quad \mathit{soldBy}(p_1, r_2) \quad \mathit{soldBy}(p_2, r_3)\ \}
\end{aligned}$$

Suppose a user request is to return all objects that have a price less than 100 with the prices and the names of the resellers. The concrete queries for this user request in the three query languages are irrelevant, since only the semantics is compared, i.e., the query results. It is easy to see the following objects are returned, together with the requested information represented w.l.o.g. by DL concepts:

$$\begin{aligned}
p_1 :& \mathit{price} = 50 \sqcap \exists \mathit{soldBy}.(\mathit{name} = \text{“}FS\text{”}) \\
& \qquad\qquad \sqcap \exists \mathit{soldBy}.(\mathit{name} = \text{“}BB\text{”}) \\
p_2 :& \mathit{price} = 60 \sqcap \exists \mathit{soldBy}.\top \\
p_3 :& \mathit{price} = 70 \sqcap \top \\
p_4 :& \mathit{price} = 50 \sqcap \exists \mathit{soldBy}.(\mathit{name} = \text{“}AZ\text{”}),
\end{aligned}$$

in which the concept $\top$ serves the purpose of `null`. □

Recall that Figure 3.1 identifies five types of queries that have specific semantics for computing query answers that are not available in the others. To give example queries for these areas, the four query answers given in Example 3 are investigated.

For p_1, the description forms a nested relational view of the resellers. An AQ query that formulates the user request in Example 3 can compute exactly the same answer, so are an corresponding SPARQL/CQ query, though the nested relation is often flattened in SPARQL/CQ, e.g., by repeating the object identifier and price information for the second reseller.

The second and third answers, p_2 and p_3, can be computed by AQ. Indeed, the reseller information given in these two query answers are different: p_2 has a reseller, whose name is unknown, while p_3 does not have any reseller. However, SPARQL cannot distinguish the two cases regarding resellers, and, in general, SPARQL would compute $p_2 : \mathit{price} = 60 \sqcap \top$ for p_2 as well. Because CQ does not have left-outer join operators, it does not compute the reseller information either p_2 or p_3. Therefore, this is an example query for the area (1) in Figure 3.1, i.e., AQ computes query answers that are unavailable in either SPARQL or CQ. In addition, it also follows that SPARQL and AQ can exploit left-outer join to compute answers that similar to p_3, while CQ cannot obtain these answers. So, similar queries that relies on left-outer joins correspond to the area (2) in Figure 3.1.

The last query answer, p_4, has its reseller information derived from the TBox: the reseller for p_4 is derived from the axiom in the TBox, and this reseller is an anonymous instance, with its name, AZ, given by the axiom as well. This is a binding of the query variable to an *existential* object. AQ and CQ are able to obtain the reseller's name for p_4, yet SPARQL would fail because binding query variables to existential instances is not supported in the current implementation. So, any queries that relies on bindings to existential instances correspond to the area (4) in Figure 3.1.

There are two more areas in Figure 3.1 that have been discussed, i.e., (3) and (5). Clearly, all instance queries reside in area (3). Area (5) indicates the queries that cannot be expressed by AQ. Since assertion queries are free of variables, hence, any SPARQL or conjunctive queries that uses more than one free variables fall in this category. For instance, consider the query that asks for *arbitrary pairs of products*.

To summarize, CQ, a subset of FOL queries, and SPARQL queries are more expressive than AQ. SPARQL queries are the most expressive among the three, for instance, variables in SPARQL queries can bind to class or role names. However, the semantics of SPARQL queries does not support existential variables, which bind to any domain element that are inferred to exist. Note that blank nodes (and variables) in SPARQL queries can only bind to *explicitly given* resources, e.g., given blank nodes or resources with an IRI. CQ, on the other hand, does not have left-outer join operators and thus is unable to compute some query answers, such as p_2 and p_3 in Example 3. Assertion queries, as a form of object queries, are easy to formulate and support both existential bindings and left-outer joins.

3.4 Computing Projection

This section provides the procedures for computing queries in the form of (C, Pd), as presented in [Pound et al., 2009]. Ultimately, computing a user query $Q = (C, Pd)$ involves computing the projection part (i.e., Pd) by the query semantics given in Definition 17. Note that Pd can be of the form $\exists R.Pd_1$, thus computing such queries require the notion of a general role path Rp, introduced as follows.

Definition 18. ***General Role Path***. Let S and D be roles and $\mathfrak{L}$-concepts for some DL fragment $\mathfrak{L}$. A *general role path* Rp is defined by the grammar:

$$Rp \quad ::= \quad Id \;\mid\; Rp.S \;\mid\; Rp.D$$

The expression $\exists Rp.D$ denotes the concept D when $Rp =$ "Id", the concept $\exists Rp_1.\exists R.D$ when $Rp =$ "$Rp_1.R$", and the concept $\exists Rp_1.(D_1 \sqcap D)$ when $Rp =$ "$Rp_1.D_1$". $\square$

With the use of a role path Rp and knowledge base $\mathcal{K}$, a top-level procedure is given by Algorithm 1. We overload the procedure name, i.e., Proj$(Pd, \mathcal{K}, Rp, a : C)$, to denote the set of *concepts* obtained by it.

Algorithm 1: Proj$(Pd, \mathcal{K}, Rp, a : C)$

```
1 switch $Pd$ do
2     case $C_1$? return ProjConcept$(C_1, \mathcal{K}, Rp, a : C)$ case $f$? return
      ProjFeature$(f, \mathcal{K}, Rp, a : C)$  case $Pd_1 \sqcap Pd_2$ return
      ProjJoin$(Pd_1, Pd_2, \mathcal{K}, Rp, a : C)$  case $\exists R.Pd_1$ return
      ProjRole$(Pd_1, R, \mathcal{K}, Rp, a : C)$
```

A straightforward coding for the procedures invoked by Proj to handle each of the four cases may lead to an unacceptably large number of tests on logical consequence. While these performance issues are unavoidable in the worst case, it is possible to improve the performance of projection computation for many situations. The subsequent sections explain how each case in Algorithm 1 is computed and optimized.

3.4.1 Procedures ProjConcepts and ProjFeature

A straightforward implementation of ProjConcept is described in Algorithm 2.

ProjFeature may be implemented analogously, shown in Algorithm 3. Observe that the function GetAllConstants$(C, \mathcal{K})$ returns the collection of all constants occurring in the knowledge bases. This function, however, can practically return constants only in the range of the feature f.

Algorithm 4 uses binary search over the set of sorted constants to efficiently compute a value for some feature. It proceeds by a progressive refinement of intervals over the

Algorithm 2: ProjConcept($C_1, \mathcal{K}, Rp, a : C$)

```
1 $result \leftarrow \emptyset$
2 if $\mathcal{K} \cup \{a : C\} \models \{a : \exists Rp.C_1\}$ then
3     $result \leftarrow result \cup \{C_1\}$
4 return $result$
```

Algorithm 3: ProjFeature($f, \mathcal{K}, Rp, a : C$)

```
1 $result \leftarrow \emptyset$
2 $constants \leftarrow$ Sort(GetAllConstants($C$, $\mathcal{K}$))
3 $low \leftarrow 0$
4 $high \leftarrow$ Size($constants$) $- 1$
5 $result \leftarrow result \cup$ IterativeSearch($f, \mathcal{K}, Rp, a, C, low, high, constants$)
6 return Reduce($result, \mathcal{K}, Rp$)
```

concrete domain, narrowing by binary search on intervals to the required set of concepts of the form $(f = k)$.

Algorithm 4: IterativeSearch($f, \mathcal{K}, Rp, a : C, low, high, constants$)

```
1  $result \leftarrow \emptyset$
2  $mid \leftarrow \lfloor (low + high)/2 \rfloor$
3  $C_1 \leftarrow (f \geq constants.$Get$(low)) \sqcap (f \leq constants.$Get$(high))$
4  if $\mathcal{K} \cup \{a : C\} \models a : \exists Rp.C_1$ then
5      if $low = high$ then
6          $result \leftarrow (f = constants.$Get$(low))$
7      else
8          $result \leftarrow result \cup$ IterativeSearch($f, \mathcal{K}, Rp, a : C, low, mid, constants$)
9          $result \leftarrow result \cup$ IterativeSearch($f, \mathcal{K}, Rp, a : C, mid+1, high, constants$)
10 return $result$
```

3.4.2 Procedure ProjJoin

A naïvely coded ProjJoin procedure for the $Pd_1 \sqcap Pd_2$ case is problematic since it requires a number of logical consequence tests equal to the product of the number of descriptions computed by Pd_1 and Pd_2. This circumstance quickly becomes intolerable for iterated uses of this procedure, e.g., for projection descriptions of the form $f_1 \sqcap \cdots \sqcap f_n$ which abstract relational-like projections that produce descriptions of tuples.

Algorithm 5: ProjJoin($Pd_1, Pd_2, \mathcal{K}, Rp, a : C$)

```
1 $result \leftarrow \emptyset$
2 foreach $C_1 \in$ Proj$(Pd_1, \mathcal{K}, Rp, a : C)$ do
3     foreach $C_2 \in$ Proj$(Pd_2, \mathcal{K}, Rp.C_1, a : C)$ do
4         $result \leftarrow result \cup \{(C_1 \sqcap C_2)\}$
5 return $result$
```

Algorithm 5 follows a *nested loops* strategy in which concepts computed by an outer loop may be used to further qualify concepts computed by an inner loop. A simple way to accomplish this is to use the general role path that enables sideways communication of outer loop concepts. There are additional opportunities for improving the performance of Algorithm 5. For one, the procedure might explore alternative permutations of projection descriptions with the form $Pd_1 \sqcap \cdots \sqcap Pd_n$ that might result in a more efficient nesting order.

3.4.3 Procedure ProjRole

The naïve implementation of this procedure involves quantifying over all subsets of concepts computed by the evaluation of the nested projection description. However, for a given knowledge base $\mathcal{K}$, role path Rp and concept C, consider the following:

- It is less likely that $\mathcal{K} \cup \{a : C\} \models \{a : \exists Rp.(\sqcap S)\}$, for concept sets S with larger numbers of elements; and
- For any pair of concept sets S_1 and S_2, if $\mathcal{K} \cup \{a : C\} \not\models \{a : \exists Rp.(\sqcap S_1)\}$, then $\mathcal{K} \cup \{a : C\} \not\models \{a : \exists Rp.(\sqcap (S_1 \cup S_2))\}$.

These observations motivate the implementation of ProjRole given by Algorithm 6 that uses a queue to conduct a breadth-first-search of subsets. Note that, to avoid considering more than one permutation of a candidate, the procedure assumes a total lexicographic order "$\prec$" over all concepts.

The algorithm works particularly well in cases where the argument projection description parameter Pd has the form $f? \sqcap Pd$, e.g., computes relational-like tuples. In this case, the number of entailment checks performed directly by the algorithm is quadratic in the size of S_1.

Algorithm 6: ProjRole($Pd, R, \mathcal{K}, Rp, a : C$)

```
1  $S_1 \leftarrow$ Project($Pd, \mathcal{K}, Rp.R, a : C$)
2  if $S_1 = \emptyset$ then
3      if $\mathcal{K} \cup \{a : C\} \models \{a : \exists(Rp.R).\top\}$ then
4          return $\{\exists(Rp.R).\top\}$
5      return $\emptyset$
6  $result \leftarrow \emptyset$
7  $queue \leftarrow \emptyset$
8  foreach $C' \in S_1$ do
9      $queue$.Enqueue($\{C'\}$)
10 while $queue$.NotEmpty() do
11     $S_2 \leftarrow queue$.Dequeue()
12     $notgrown \leftarrow true$
13     foreach $C' \in S_1$ where $\forall C'' \in S_2 : C'' \prec C'$ do
14         if $\mathcal{K} \cup \{a : C\} \models \{a : \exists(Rp.R).(C' \sqcap (\sqcap S_2))\}$ then
15             $queue$.Enqueue($\{C'\} \cup S_2$)
16             $notgrown \leftarrow false$
17     if $notgrown$ then
18         $result \leftarrow result \cup \{\exists R.(\sqcap S_2)\}$
19 return Reduce($result, \mathcal{K}, Rp$)
```

3.4.4 Procedure Reduce

The procedure Reduce ensures the most informative concepts, as defined in Definition 16, are selected from the set of concepts before returning results from previous procedures. An implementation of this procedure is given by Algorithm 7.

Algorithm 7: Reduce($S, \mathcal{K}, Rp$)

```
1  $result \leftarrow \emptyset$
2  foreach $C' \in S$ do
3      $minimal \leftarrow true$
4      foreach $C'' \in S - \{C'\}$ do
5          if $\mathcal{K} \models \exists Rp.C'' \sqsubseteq \exists Rp.C'$ and $\mathcal{K} \not\models \exists Rp.C' \sqsubseteq \exists Rp.C''$ then
6              $minimal \leftarrow false$
7              break
8      if $minimal$ then
9          $result \leftarrow result \cup \{C'\}$
10 if $\mathcal{K} = \emptyset$ then
11     return $result$
12 else
13     return Reduce($result$, $\emptyset$, $Rp$)
```

3.4.5 Correctness

Recall that $\llbracket S \rrbracket_{\mathcal{K}}$ is not presumed a singleton set, which means that a total ordering of all concepts in the corresponding DL dialect $\mathfrak{L}$ is needed. The procedures invoked in Algorithm 1 assume a simple lexicographical order on all concepts. In particular, relevant symbols adhere to the following order:

$$\neg \prec \exists^{\leq n} \prec \sqcap \prec f = k \prec A \prec \top,$$

where f is a concrete feature, k a constant, and $A \in$ NC. Also, when Proj$(Pd, \mathcal{K}, Rp, a : C) = \emptyset$, $\bigsqcap$Proj$(Pd, \mathcal{K}, Rp, a : C) = \top$.

Theorem 3.4.1. *For any query* $Q = (C, Pd)$, $\textsc{Eval}(Q, \mathcal{K}) = \{a : \bigsqcap \textsc{Proj}(Pd, \mathcal{K}, Rp, a : C) \mid \mathcal{K} \models a : C\}$ *w.r.t. the given lexicographical ordering on concepts, where* $\textsc{Proj}(\cdot)$ *is given by Algorithm 1.*

Proof. To prove the correctness of Algorithm 1, first observe that it suffices to prove each of the four procedures used in computing each case of Pd shown in Algorithm 1. Recall the query semantics given in Definition 17, it is sufficient to show, for each algorithm computing a case of Pd, $\bigsqcap \textsc{Proj}(Pd, \mathcal{K}, Rp, a : C) = \llfloor \{D \mid D \in \mathcal{L}_{Pd}, \mathcal{K} \models a : D\} \rrfloor_{\mathcal{K}}$, where $\mathcal{K} \models a : C$. To ease the presentation, in the following proofs use S_{Pd} to denote the set $\{D \mid D \in \mathcal{L}_{Pd}, \mathcal{K} \models a : D\}$.

Since Algorithm 7 is used by other algorithms, it is proven first. The purpose of this algorithm is to simulate $\llfloor S \rrfloor_{\mathcal{K}}$ for some set of concepts S, which includes two reductions: one w.r.t. $\mathcal{K}$ and the other w.r.t. $\emptyset$. The loop in Line 2 iterates all concepts in S, and, for each concept C', another loop in Line 4 tests, for all other concepts in S, if the condition in Line 5 is met. This is exactly the condition in Definition 16 for finding the most informative concepts. If the condition is met, then C' is not one of the most informative concept (thus C' is not one of the answers). If the condition is not satisfied, then C' is indeed an answer. It is easy to see that both loops terminate because of the flag variable *minimal* and the finiteness of S. The second reduction just calls the algorithm itself, except that $\emptyset$ is used instead of $\mathcal{K}$. After computing the most informative concepts, the algorithm terminates because of Line 11. After the second reduction, $result = \llfloor S \rrfloor_{\mathcal{K}}$.

Algorithm 2. By Definition 14, when $Pd = C_1?$, $\mathcal{L}_{Pd} = \{C_1, \top, C_1 \sqcap \top\}$. If the condition in Line 2 is satisfied (note $Rp = Id$), then $C_1 \in \mathcal{L}_{Pd}$ and $\mathcal{K} \models a : C_1$. So, C_1 is returned. On the other hand, $S_{Pd} = \{C_1, C_1 \sqcap \top\}$. In this case, although $C_1 \equiv C_1 \sqcap \top$, the concept $C_1 \sqcap \top$ will not be selected due to the lexicographical ordering imposed and it is dismissed; therefore, $\llfloor S_{Pd} \rrfloor_{\mathcal{K}} = C_1$. If the condition in Line 2 is not satisfied, then $result = \emptyset$, and $\top$ is returned. On the other hand, because $\mathcal{K} \not\models a : C_1$, $S = \{\top\}$ and $\llfloor S_{Pd} \rrfloor_{\mathcal{K}} = \top$. Hence, this algorithm is correct w.r.t. the aforementioned ordering.

Algorithm 3. First, *low* and *high* store the lowest and highest indices of all the constants, respectively. Note that the list of constants is necessarily finite, considering a given knowledge base, and it is sorted. Then, an iterative search procedure, Algorithms 4, is invoked to find all valid results. In Algorithms 4, the index of constant located in the middle of the list is computed. Line 4 then checks if the potential answers are within the range of the low and high constants. If the condition is true, then the search proceeds in a binary fashion. Because of this straightforward binary search, the recursive calls to this procedure will terminate when the high index equals the low one. Now consider the case $Rp = Id$. If the condition in Line 4 is satisfied, only one answer is returned, say

$f = k$, since otherwise $\mathcal{K}$ is inconsistent. Hence, $f = k$ is returned (Algorithm 7 does not change the result for a singleton set). If the condition in Line 4 is not met, $\top$ is returned. Hence, for the case $Rp = Id$, the proof is analogous to Algorithm 2. The case $Rp \neq Id$ is considered in Algorithm 6.

Algorithm 5. This procedure is a straightforward application of Lemma 3.2.2. Observe that the reduction step is not needed for the results in this procedure, because Lemma 3.2.2 ensures the results are the concepts that would be retained after reductions. In addition, the order of the selected answers is preserved by induction on Pd_1 and Pd_2.

Algorithm 6. This procedure first computes computes $\mathcal{L}^{\text{TUP}}_{Pd}$ for the sub projection description Pd, and stores the results in S_1, which follows from Definition 14. When S_1 is empty, the procedure checks if "$\exists(Rp.R).\top$" is an answer. The proof for this part is analogous to that of Algorithm 2, assuming the given lexicographical order. When S_1 is non-empty, each concept in this set is made a singleton set and stored in *queue*, i.e., Line 9. The rest of this procedure attempts to add more concepts to the singleton sets to make them more specific. Specifically, the loop guard at Line 13 ensures that a concept that is ordered after every concept in the set S_2 is processed, thus ensuring the termination of this loop (otherwise S_2 can keep adding the same concept in itself and Line 14 remains true). Line 14 guarantees that the concept C' is added to S_2 only if it still satisfies the condition C while making S_2 more specific. The outer while loop will eventually terminate because the termination of the inner for loop will not set the flag *notgrown*, thus stopping the augmentation of the queue. When *notgrown* is true, S_2 is saturated and no more concept can make it more specific, and it is an answer. It is easy to see the procedure computes all combination of concepts in S_1 in a power set fashion, i.e., the queue contains all sets of size 1, of size 2, and so on. So, the procedure computes a subset of $\mathcal{L}_{\exists(Rp.R).Pd}$, in which each concept is the most specific w.r.t. the given ordering for a particular permutation of concepts in S_1, e.g., when the answer set has $\exists(Rp.R).(C_1 \sqcap C_2)$, it would not contain $\exists(Rp.R).C_1$ or $\exists(Rp.R).C_2$. The results are reduced subsequently, i.e., it computes exactly the set of concepts for $\lfloor\lfloor S_{Pd} \rfloor\rfloor_{\mathcal{K}}$. Since this set is not necessarily singleton, a conjunction of its elements would equal $\lfloor\lfloor S_{Pd} \rfloor\rfloor_{\mathcal{K}}$. □

3.5 A Query Plan Language

When evaluating queries over relational databases, a relational DBMS performs query optimization to enhance efficiency, as discussed in Section 5.1. Typically, a DBMS will use relational algebra to generate and rewrite query plans in order to find an efficient plan for query execution. The same strategy is adopted for query evaluation over knowledge bases.

This section defines a *query algebra* for manipulating sets of concept assertions that functions as the query plan language. This algebra can be used to describe a variety of query plans that can vary widely in the cost of their evaluation for evaluating a user query Q in Definition 17. These query plans, or algebraic expressions, are considered to be transparent to end users, who can only pose user queries. To distinguish a user query from a query plan, the latter (an algebraic expression) is denoted $\mathtt{Q}$. Thus, computing a user query Q is reduced to evaluating some query plan $\mathtt{Q}$. Since evaluating a query plan $\mathtt{Q}$ may require additional resources, such as the underlying knowledge base $\mathcal{K}$ and/or a set of cached query results (see Definition 21) $\mathbb{SI}$, the evaluation of $\mathtt{Q}$ is denoted $[\![\mathtt{Q}]\!]_{\mathcal{K}}^{\mathbb{SI}}$. The goal of query optimization is to find an *efficient* query plan $\mathtt{Q}$ for a user query Q such that $\textsc{Eval}(Q, \mathcal{K}) = [\![\mathtt{Q}]\!]_{\mathcal{K}}^{\mathbb{SI}}$.

It is important that the set of assertions to be manipulated by algebraic operations be organized in some way for efficient data access. For this purpose, an implicit ordering can be imposed on a set of assertions, and such an ordered set can be arranged by a special data structure called *description indices*. In the following, Section 3.5.1 defines how an ordering is used in description indices, which are in turn exploited by algebraic operators defined in Section 3.5.2.

3.5.1 Description Index

A description index is a search tree in which nodes correspond to concepts and in which search order is defined by an *ordering description* (or Od for short) [Pound et al., 2008]. In this work, description trees are extended to support efficient search for concept assertions; the following ordering descriptions are considered:

Definition 19. ***Ordering Description***. Let C be an $\mathfrak{L}$ concept for some DL dialect $\mathfrak{L}$, C^* a concept obtained from C by replacing all f occurring in C by f^*, R by R^*, and A by A^*, where $f \in \text{NF}$, $R \in \text{NR}$ and $A \in \text{NC}$. An ordering description Od is defined by the following grammar:

$$Od \ ::= \ \text{Un} \ \mid \ \text{Ind} \ \mid \ f : Od_1 \ \mid \ [C](Od_1, Od_2),$$

where an instance of each production is called *null*, *identifier*, *feature*, and *partition* ordering, respectively. Given two concept assertions $a_1 : C_1$ and $a_2 : C_2$, the former *precedes* the latter w.r.t. the ordering description Od, denoted $\prec_{Od} (a_1 : C_1, a_2 : C_2)$, if at least one of the following conditions hold:

- $Od = \text{Ind}$ and $a_1 \prec a_2$, where $\prec$ denotes the usual lexicographical order;

- $Od = f : Od_1$ and $\models C_1 \sqcap C_2^* \sqsubseteq (f < f^*)$;
- $Od = f : Od_1$, $\models C_1 \sqcap C_2^* \sqsubseteq (f = f^*)$, and $\prec_{Od_1} (a_1 : C_1, a_2 : C_2)$;
- $Od = [C](Od_1, Od_2)$, $\models C_1 \sqsubseteq C$, and $\not\models C_2 \sqsubseteq C$;
- $Od = [C](Od_1, Od_2)$, $\models C_1 \sqsubseteq C$, $\models C_2 \sqsubseteq C$, and $\prec_{Od_1} (a_1 : C_1, a_2 : C_2)$;
- $Od = [C](Od_1, Od_2)$, $\not\models C_1 \sqsubseteq C$, $\not\models C_2 \sqsubseteq C$, and $\prec_{Od_2} (a_1 : C_1, a_2 : C_2)$.

Two concept assertions $a_1 : C_1$ and $a_2 : C_2$ are *incomparable* w.r.t. Od if neither $\prec_{Od} (a_1 : C_1, a_2 : C_2)$ nor $\prec_{Od} (a_2 : C_2, a_1 : C_1)$ hold. □

Properties of description indices presented in [Pound et al., 2008] are still valid for the above definition, while the notion of *descriptive sufficiency* deserves extra attention, as defined below:

Definition 20. ***Descriptive Sufficiency***. A concept assertion $a : C$ is sufficiently descriptive w.r.t. Od, denoted $\mathrm{SD}(a : C, Od)$, if any one of the following conditions hold:

- $Od = \mathrm{Un}$;
- $Od = \mathrm{Ind}$;
- $Od = f : Od_1$, $\models C \sqsubseteq (f = s)$ for some $s \in \mathbb{D}$, and $\mathrm{SD}(a : C, Od_1)$.

□

Intuitively, descriptive sufficiency shows the "completeness" of a concept assertion w.r.t. the given Od. Such information is particularly useful for searching over a set of concept assertions. The use of descriptive sufficiency will be revisited in Chapter 5.

For description indices arranged w.r.t certain ordering description, searching for particular concept assertion(s) can be very efficient. The following examples are given to illustrate the uses of Od:

Example 4. ***Feature and Identifier Ordering*** A collection of products with price information is indexed by a description index; the concept assertions are additionally indexed by the ordering description *price* : Ind, i.e., with a major sort on *price* and a minor sort on individual names (identifiers). Figure 3.2 illustrate such a description index with five products. Clearly, an in-order traversal of this description index returns the products

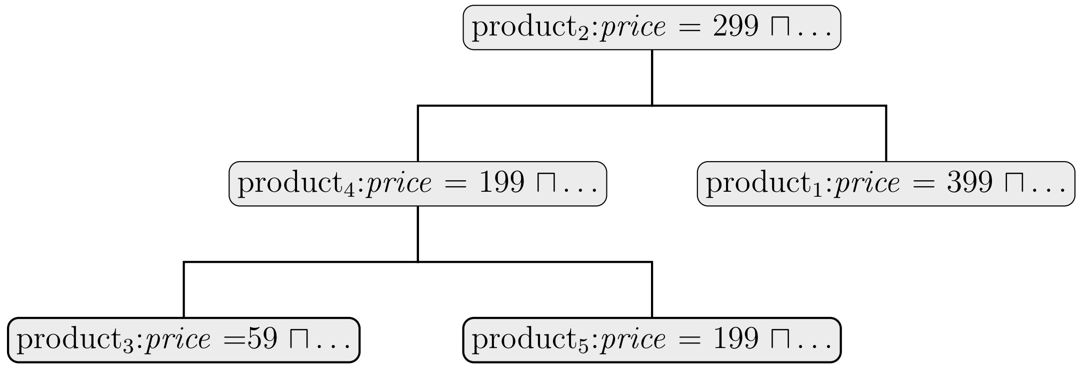

product$_2$:*price* = 299 ⊓ . . .
product$_4$:*price* = 199 ⊓ . . .
product$_1$:*price* = 399 ⊓ . . .
product$_3$:*price* =59 ⊓ . . .
product$_5$:*price* = 199 ⊓ . . .

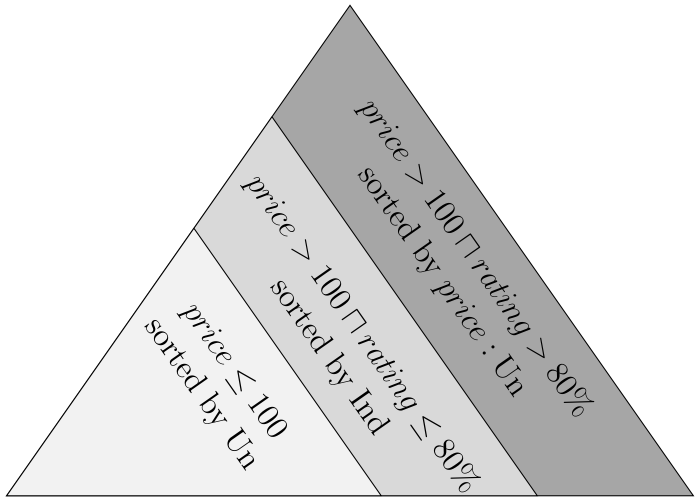
Figure 3.2: A description index with $Od = price : \text{Ind}$.

in a non-descending order of product prices; in addition, when the major sort of *Price* fails, the minor sort on individual names is effective. □

In Example 4, a query on the major sort feature (*price*) can be computed efficiently over the description index. However, such good performance is only possible for one-dimension search, i.e., the major sort feature. In reality, multidimensional queries are not uncommon, which often require some spatial data structure; for instance, web users may want to search products by price and rating ranges. Such requests can also be accommodated by a description index with partition ordering, as shown in Example 5.

Example 5. ***Partition Ordering.*** A collection of products with price and user rating information is indexed by a description index with the ordering description $[price > 100]([rating > 80\%](price : \text{Un}, \text{Ind}), \text{Un})$. The ordering description can be understood as follows. First, partition products into two groups by checking if the product price is over 100. The more expensive products (group 1) is further partitioned into two smaller ones (groups 1.1 and 1.2) by checking if the user rating of a product is over 80%, and the well-rated products (group 1.1) are sorted by price, while the rest (group 1.2) are sorted by product identifiers. For the less expensive products (group 2), products need not to be sorted. A more intuitive explanation is given in Figure 3.3, which abstracts the index as a tree and partitions as parts of the tree. □

Starting from the left-most partition in Figure 3.3, the shaded parts correspond to the collections of products in group 2, group 1.2, group 1.1, respectively. Indeed, partition ordering can be used by description indices to achieve the behaviour of *k-d trees* and to facilitate multidimensional search. Ordering description is even more flexible for this purpose; for instance, the index in Example 5 does not alternate between the two dimensions



$price > 100 \sqcap rating > 80\%$
sorted by $price$: Un
$price > 100 \sqcap rating \leq 80\%$
sorted by Ind
$price \leq 100$
sorted by Un

Figure 3.3: A description index with a partition ordering.

as a *k-d* tree would.

3.5.2 The Algebra

Six algebraic operators have been defined in the query algebra, as shown in Table 3.1. Before introducing the semantics of these algebraic operators, a *cached query result* is formally defined as follows:

Definition 21. ***Cached Query Result***. A *cached query result*, $S_i = Q :: Od$, is the set of concept assertions obtained by evaluating the user query $Q = (C_i, Pd_i)$ over some knowledge base $\mathcal{K}$, i.e., $\textsc{Eval}(Q, \mathcal{K})$, which are stored in a description index with respect to some ordering Od. □

As the name suggests, a cached query result stores the set of concept assertions obtained by evaluating a user query and the materialized results are indexed by a description index to facilitate search. When clear from the context, a cached query result is also called a cached query. Relating to relational databases, a cached query can be thought of as a *materialized view*. However, most current DL reasoners do not support result caching [Weithöner et al., 2006], hence, these DL systems cannot benefit from cached results even though a user query is repetitively posed.

Query plans can exploit cached query results to facilitate efficient search over a set of concept assertions. Definition 21 permits the existence of any number of cached query results often organized in data structures designed to support user queries. This is analogous to relational systems, where multiple specialized indices are defined to support queries. Cached query results are essential in enhancing the performance of actual implementations because they enable query evaluation to reduce or even avoid general DL reasoning, as discussed in Section 5.2.

With the notion of cached query results in Definition 21, the assertion retrieval algebra can now be defined.

Definition 22. ***Algebraic Operators***. An algebraic operator $\mathtt{Q}$ is defined as follows:

$$\mathtt{Q} := \ C \ \mid \ P^{\mathcal{K}} \ \mid \ S_i(\mathtt{Q}) \ \mid \ \sigma_C^{\mathcal{K}}(\mathtt{Q}) \ \mid \ \pi_{Pd}^{\mathcal{K}}(\mathtt{Q}) \ \mid \ \mathtt{Q} \bowtie \mathtt{Q}.$$

$\square$

A query plan is then formed by a sequence of algebraic operations, i.e., an algebraic expression. The semantics of each operator is given in Table 3.1. Note that each operator maps sets of assertions to a set of assertions. In particular, the set of assertions obtained by an evaluation of a query plan $\mathtt{Q}$ is denoted $[\![\mathtt{Q}]\!]_{\mathcal{K}}^{\mathbb{SI}}$, where $\mathbb{SI}$ is a set of supplied cached query results and $\mathcal{K}$ some knowledge base.

The *constant* operator C is a leaf operator, defined to be a set of concept assertions in which the concept part is the supplied C. Without loss of generality, the result can be compactly represented by a single generalized assertion of the form $\{\star : C\}$ where $\star$ stands for an arbitrary individual in $\mathcal{K}$.

The *primary* operator $P^{\mathcal{K}}$, an alternative to the user query $(\top, \top?)$, is also a leaf operator that obtains all the instance names in the underlying $\mathcal{K}$, without sorting assertions. As discussed in Chapter 1, no *standard design* is implemented in knowledge bases, so, not every user query has a default query plan as in the relational setting. The operator $P^{\mathcal{K}}$ circumvents the standard design problem by providing a mechanism to return a concept assertion for every individual that appears in the knowledge base. In terms of physical design, the set of assertions computed by $P^{\mathcal{K}}$ can be supported by any data structure, particularly, a description index (see Section 3.5.1).

The *cache scan* $S_i(\mathtt{Q})$ links the algebra to the underlying *cached query results*. While not mandated by the definitions, the subquery $\mathtt{Q}$ of a given $S_i(\mathtt{Q})$ operator is also expected to be related to the specification of the underlying cached query result S_i to facilitate efficient search. Also note the run-time optimization of compactly representing the results of a constant operator can lead to improved efficiency when executing $S_i(C)$.

Operator $\mathtt{Q}$	Semantics $[\![\mathtt{Q}]\!]^{\mathbb{SI}}_{\mathcal{K}}$
C	$\{\star : C \mid \star$ stands for any arbitrary individual$\}$
$P^{\mathcal{K}}$	$\{a : \top \mid a$ appears in $\mathcal{K}\}$
$S_i(\mathtt{Q})$	$\{a : C \mid (a : C) \in S_i, \exists(a : D) \in \mathtt{Q}, \{a : C\} \models a : D\} \cup$ $\{a : C \mid (a : C) \in S_i, \{a : C\} \models a : D, \{\star : D\} = \mathtt{Q}\}$
$\sigma^{\mathcal{K}}_{C}(\mathtt{Q})$	$\{a : D \mid \exists(a : D) \in \mathtt{Q}, \mathcal{K} \cup \{a : D\} \models a : C\} \cup$ $\{a : D \mid \mathcal{K} \cup \{a : D\} \models a : C, \{\star : D\} = \mathtt{Q}\}$
$\pi^{\mathcal{K}}_{Pd}(\mathtt{Q})$	$\{a : \lfloor\!\lfloor\{D \mid \mathcal{K} \cup \{a : C\} \models a : D, D \in \mathcal{L}_{Pd}\}\rfloor\!\rfloor_{\mathcal{K}} \mid \exists(a : C) \in \mathtt{Q}\} \ \cup$ $\{\star : \lfloor\!\lfloor\{D \mid \mathcal{K} \models C \sqsubseteq D, D \in \mathcal{L}_{Pd}\}\rfloor\!\rfloor_{\mathcal{K}}, \{\star : D\} = \mathtt{Q}\}$
$\mathtt{Q}_1 \bowtie \mathtt{Q}_2$	$\{a : D_1 \sqcap D_2 \mid \exists(a : D_i) \in \mathtt{Q}_i, i = 1, 2\} \ \cup$ $\{a : D_1 \sqcap D_2 \mid \{\star : D_1\} = \mathtt{Q}_1, \exists(a : D_2) \in \mathtt{Q}_2\} \cup$ $\{a : D_1 \sqcap D_2 \mid \exists(a : D_1) \in \mathtt{Q}_1, \{\star : D_2\} = \mathtt{Q}_2\} \ \cup$ $\{\star : D_1 \sqcap D_2 \mid \{\star : D_1\} = \mathtt{Q}_1, \{\star : D_2\} = \mathtt{Q}_2\}$

Table 3.1: Algebraic operators and the semantics.

The *selection* operator $\sigma^{\mathcal{K}}_{C}(\mathtt{Q})$ and the *projection* operator $\pi^{\mathcal{K}}_{Pd}(\mathtt{Q})$ are similar to the corresponding relational algebraic operators, except that general DL reasoning may be necessary. The *join* operator $\mathtt{Q}_1 \bowtie \mathtt{Q}_2$ is, from the perspective of relational algebra, more a natural join of two set of concept assertion, by merging the concept descriptions of common instances (identified by names) occurring in the result sets of both sub-operators.

Definition 23. ***Pure and Impure Operators***. An algebraic operator $\mathtt{Q}$ is either *pure* or *impure*: C is impure, $P^{\mathcal{K}}$ and $S_i(Q)$ are pure, $\sigma^{\mathcal{K}}_{C}(\mathtt{Q})$ and $\pi^{\mathcal{K}}_{Pd}(\mathtt{Q})$ are pure if $\mathtt{Q}$ is pure, and $\mathtt{Q}_1 \bowtie \mathtt{Q}_2$ are pure if both $\mathtt{Q}_1$ and $\mathtt{Q}_2$ are pure. □

Intuitively, for a consistent knowledge base $\mathcal{K}$, a pure operator only generates consistent concept assertions, while an impure one could potentially generate concept assertions that are inconsistent. This property is captured by the following lemma:

Lemma 3.5.1. *For any consistent $\mathcal{K}$ and any* pure *$\mathtt{Q}$, if $a : D \in [\![\mathtt{Q}]\!]^{\mathbb{SI}}_{\mathcal{K}}$, then $\mathcal{K} \cup \{a : D\}$ is also consistent.*

Proof. The proof is by structural induction on $\mathtt{Q}$.

- When $\mathtt{Q} = P^{\mathcal{K}}$, if $a : D \in [\![\mathtt{Q}]\!]^{\mathbb{SI}}_{\mathcal{K}}$, then $D = \top$ and $\mathcal{K} \cup \{a : \top\}$ must be consistent assuming the consistency of $\mathcal{K}$.

- When $\mathtt{Q} = S_i(\mathtt{Q}')$, if $a : D \in [\![\mathtt{Q}]\!]^{\mathbb{SI}}_{\mathcal{K}}$, then $a : D \in S_i$, which implies $\mathcal{K} \models a : D$ by Definition 17. So, $\mathcal{K} \cup \{a : D\}$ must be consistent.

- When $\mathtt{Q} = \sigma^{\mathcal{K}}_{C}(\mathtt{Q}')$ and $\mathtt{Q}'$ is also pure, if $a : D \in [\![\mathtt{Q}]\!]^{\mathbb{SI}}_{\mathcal{K}}$, then if $a : D \in [\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}}$, which, by induction hypothesis on pure $\mathtt{Q}'$, shows that $\mathcal{K} \cup \{a : D\}$ is consistent.

- When $\mathtt{Q} = \pi^{\mathcal{K}}_{Pd}(\mathtt{Q}')$ and $\mathtt{Q}'$ is also pure. If $a : D \in [\![\mathtt{Q}]\!]^{\mathbb{SI}}_{\mathcal{K}}$, then there is $a : D' \in [\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}}$ such that $\mathcal{K} \cup \{a : D'\} \models a : D$. By induction hypothesis, $\mathcal{K} \cup \{a : D'\}$ must be consistent because $\mathtt{Q}'$ is pure, so $\mathcal{K} \cup \{a : D\}$ is also consistent, as otherwise $\mathcal{K} \cup \{a : D'\} \cup \{a : D\}$ is inconsistent.

- When $\mathtt{Q} = \mathtt{Q}_1 \bowtie \mathtt{Q}_2$ and $\mathtt{Q}_1$, $\mathtt{Q}_2$ are both pure. If $a : D \in [\![\mathtt{Q}]\!]^{\mathbb{SI}}_{\mathcal{K}}$, where $D \equiv D_1 \sqcap D_2$ and $a : D_i \in [\![\mathtt{Q}_i]\!]^{\mathbb{SI}}_{\mathcal{K}}, i \in \{1, 2\}$. It is easy to see that $\mathcal{K} \cup \{a : D\}$ is consistent by induction hypotheses on two pure queries $\mathtt{Q}_1$ and $\mathtt{Q}_2$.

$\square$

Given the assertion retrieval algebra, a user query can *always* be translated into an algebraic expression that serves as starting point for the subsequent query optimization discussed in Chapter 5. This property is stated formally in Theorem 3.5.2.

Theorem 3.5.2. *A user query (C, Pd) can always be expressed by the algebraic expression*

$$\pi^{\mathcal{K}}_{Pd}(\sigma^{\mathcal{K}}_{C}(P^{\mathcal{K}})) \tag{3.1}$$

Proof. Let $Q = (C, Pd)$ and $\mathtt{Q} = \pi^{\mathcal{K}}_{Pd}(\sigma^{\mathcal{K}}_{C}(P^{\mathcal{K}}))$, we show that $\textsc{Eval}(Q, \mathcal{K}) = [\![\mathtt{Q}]\!]^{\mathbb{SI}}_{\mathcal{K}}$. Observe that $[\![P^{\mathcal{K}}]\!]^{\mathbb{SI}}_{\mathcal{K}} = \{a : \top \mid a \text{ appears in } \mathcal{K}\}$, hence $[\![\sigma^{\mathcal{K}}_{C}(P^{\mathcal{K}})]\!]^{\mathbb{SI}}_{\mathcal{K}} = \{a : \top \mid \mathcal{K} \models a : C \text{ and } a \text{ appears in } \mathcal{K}\}$ by the semantics defined in Table 3.1. In addition,

$$[\![\pi^{\mathcal{K}}_{Pd}(\sigma^{\mathcal{K}}_{C}(P^{\mathcal{K}}))]\!]^{\mathbb{SI}}_{\mathcal{K}} = \{a : \llcorner\!\!\llcorner\{D \mid D \in \mathcal{L}_{Pd}, \mathcal{K} \models a : D\}\lrcorner\!\!\lrcorner_{\mathcal{K}} \mid \mathcal{K} \models a : C \text{ and } a \text{ appears in } \mathcal{K}\} \tag{3.2}$$

by the semantics of the projection operator. It then follows by Definition 17 that $[\![\mathtt{Q}]\!]^{\mathbb{SI}}_{\mathcal{K}}$ computes the same set of concept assertions as $\textsc{Eval}(Q, \mathcal{K})$ does. $\square$

This chapter presents a query language for assertion retrieval, which generalizes instance retrieval to allow for projection descriptions. The semantics of assertion queries are based on instance checking, a basic reasoning task that instance retrieval also relies on. Hence, to answer assertion queries efficiently, optimization techniques for instance checking are necessary. As shown in Section 2.4.1, most of the existing query optimization techniques over DL knowledge bases concentrate on instance queries. In this work, we propose a novel optimization for *instance checking* in Chapter 4, which can be used to address the efficiency of answering assertion queries.

The intrinsic complexity of instance checking renders assertion retrieval impractical, despite the novel optimization developed in this work. This is because an assertion query with a non-trivial projection description could entail a large volume of instance checking requests, as can be seen from the procedures given in Section 3.4. A solution can be obtained from relational databases, that is, cost-based query optimization can be used to improve assertion retrieval. The query plan language introduced in Section 3.5 is designed for this purpose. Similar to relational algebra in the relational model, the plan language can be used to rewrite a user query into a non-empty set of algebraic expressions for cost-based query optimization. Chapter 5 elaborates how this query plan language is exploited to optimize query answering for assertion queries, with the use of description indices introduced in this chapter.

Chapter 4

ABox Absorption

A fundamental task over DL knowledge bases is to determine if they are *consistent*. As mentioned earlier, instance checking (see Definition 10) is pivotal for query answering. Usually, instance checking is assumed to include consistency checking of $\mathcal{K}$. However, typical workloads for a reasoning service will include far more instance checking tasks than knowledge base consistency tasks. In practice, most of the knowledge bases that queries are issued against are indeed consistent. Thus, the resulting "separation of concerns" can therefore enable technology that is considerably more efficient for such workloads.

This chapter introduces a novel adaptation of binary absorption [Hudek and Weddell, 2006] for DL knowledge bases and demonstrate that the technique is efficacious for instance checking. In particular, this technique is useful for situations that require a large number of instance checking tasks, such as instance queries and assertion queries, and that non-Horn DL $\mathcal{T}$ that precludes the possibility of computing pre-completion from $\mathcal{A}$ (e.g., when disjunction is used in $\mathcal{T}$) [Wu et al., 2012b,a]. This novel technique can substantially improve performance for instance checking over consistent knowledge bases because it allows for a tableaux algorithm to only explore a (potentially much smaller) subset of $\mathcal{A}$, achieved by the so-called *guarded reasoning* to be elaborated in this chapter.

To consider performance issues for instance checking in the new optimization, we first consider how one can map instance checking problems to concept satisfaction problems in which consistency is assumed, and then revisit *absorption* in this new setting. In particular, this chapter presents an absorption algorithm, *ABox absorption*, that is an adaptation of binary absorption, reported in [Hudek and Weddell, 2006].

Binary absorption combines two key ideas. The first makes it possible to avoid generating (at least some of the) disjunctions for axioms of the form $(A_1 \sqcap A_2) \sqsubseteq C$, where the

A_i denote *primitive concepts* and C a general concept. The second is an idea relating to *role absorptions* shown in [Tsarkov and Horrocks, 2004]. To illustrate, binary absorption makes it possible to completely absorb the axiom

$$A_1 \sqcap (\exists R_1^-.A_2) \sqcap (\exists R_2.(A_3 \sqcup A_4)) \sqsubseteq A_5.$$

In this case, the absorption would consist of a set of axioms with a single atomic concept on the left-hand side

$$\{A_2 \sqsubseteq \forall R_1.A_6, A_3 \sqsubseteq A_7, A_4 \sqsubseteq A_7, A_7 \sqsubseteq \forall R_2^-.A_8\}$$

and a second set of axioms with a conjunction of two primitive concepts on the left-hand side

$$\{(A_1 \sqcap A_6) \sqsubseteq A_9, (A_9 \sqcap A_8) \sqsubseteq A_5\},$$

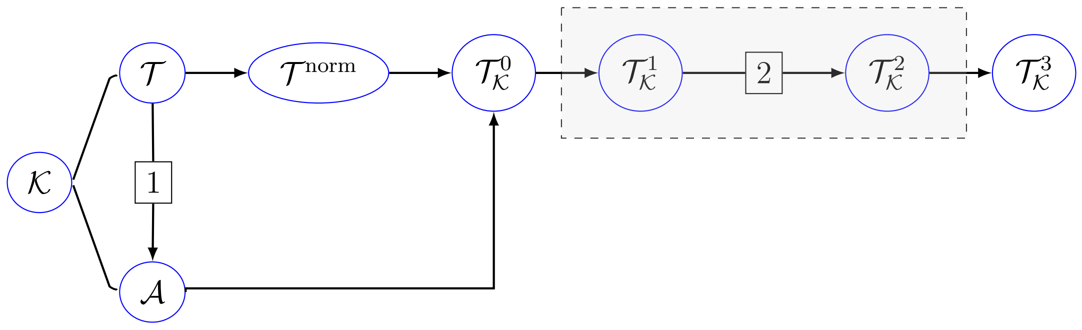
in which A_6, A_7, A_8 and A_9 are fresh atomic concepts introduced by the binary absorption procedure. Hereon, an instance of the latter set is referred to as a *binary absorption*. A key insight is that it is not necessary for *both* concepts occurring in the left-hand-side of such a dependency to be atomic. In particular, binary absorption raises the possibility of reducing instance checking problems to concept subsumption problems via the introduction of nominals in such axioms, but without suffering the consequent overhead that doing so would almost certainly entail without binary absorption.

Note that there are other reasons that binary absorption is useful, beyond the well-documented advantages of reducing the need for internalization of general terminological axioms. In particular, it works very well for the parts of a terminology that are Horn-like, as illustrated by the above example.

The approach for ABox absorption proceeds in a series of steps illustrated in Figure 4.1. An input $\mathcal{SHIQ}(\mathbb{D})$ knowledge base $\mathcal{K}$ is first separated into its constraints $\mathcal{T}$ (a TBox) and assertions $\mathcal{A}$ (an ABox). A *normalized* TBox $\mathcal{T}^{\text{norm}}$ is then obtained from $\mathcal{T}$, and a series of subsequent TBoxes, $\mathcal{T}_\mathcal{K}^i$, are derived from $\mathcal{T}^{\text{norm}}$ and the ABox $\mathcal{A}$, ultimately obtaining an absorbed $\mathcal{SHOIQ}(\mathbb{D})$ TBox $\mathcal{T}_\mathcal{K}^3$. It should be pointed out that $\mathcal{T}_\mathcal{K}^0$ can already be absorbed by the proposed absorption algorithm. Consequently, computing $\mathcal{T}_\mathcal{K}^1$ and $\mathcal{T}_\mathcal{K}^2$ in the shaded box in Figure 4.1 is not mandatory. However, as discussed in Section 4.1.2, $\mathcal{T}_\mathcal{K}^0$ can be further optimized without too much efforts such that the resulting $\mathcal{T}_\mathcal{K}^1$ and $\mathcal{T}_\mathcal{K}^2$ can again improve query performance, which has been corroborated by empirical studies presented in Section 4.4.

There are two labeled arcs in Figure 4.1 that indicate where additional processing might be useful. In particular, the arc labeled "1" is where a process called *nominal absorption* can be applied that would allow our method to be used for $\mathcal{SHOIQ}(\mathbb{D})$ knowledge bases that admit a limited use of nominals [Sirin et al., 2006]. The arc labeled "2" is where an intermediate process might be included to reduce the number of reasoning tasks required when computing $\mathcal{T}_{\mathcal{K}}^2$ using $\mathcal{T}_{\mathcal{K}}^1$. The techniques suitable for these two arcs are not considered in this work and are left as future work.

This chapter is organized as follows: Section 4.1 elaborates the mapping from instance checking problems to subsumption checking problems, which discusses the processes to obtain $\mathcal{T}_{\mathcal{K}}^i, 0 \leq i \leq 2$ from any $\mathcal{SHIQ}(\mathbb{D})$ knowledge base $\mathcal{K}$, as shown in Figure 4.1. Furthermore, the main theoretic result on the soundness and completeness of such a mapping procedure is given and proven in Section 4.1.3. Section 4.2 extends the original binary absorption to support ABox absorption, particularly, to account for nominals that are introduced in Section 4.1. Section 4.3 defines a procedure for general binary absorption that is capable of absorbing any mapped knowledge base, e.g., $\mathcal{T}_{\mathcal{K}}^i, 0 \leq i \leq 2$, into the final $\mathcal{T}_{\mathcal{K}}^3$.



$\mathcal{K}$
$\mathcal{T}$
$\mathcal{T}^{\text{norm}}$
$\mathcal{T}_{\mathcal{K}}^0$
$\mathcal{T}_{\mathcal{K}}^1$
2
$\mathcal{T}_{\mathcal{K}}^2$
$\mathcal{T}_{\mathcal{K}}^3$
1
$\mathcal{A}$

Figure 4.1: Obtaining an ABox absorption.

4.1 Obtaining an ABox Absorption

Given a $\mathcal{SHIQ}(\mathbb{D})$ knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, an ABox absorption proceeds as follows. First, *guards* are added to ABox assertions in $\mathcal{A}$. All ABox assertion are then in turn converted into TBox axioms via the use of nominals. Axioms in $\mathcal{T}$ are responsible for introducing guards that can be used to activate appropriate ABox individuals and assertions during reasoning. These processes result in $\mathcal{T}_{\mathcal{K}}^0$, as presented in Section 4.1.1. A further optimization can be applied to $\mathcal{T}_{\mathcal{K}}^0$ to relax typing constraints in the form of $L_1 \sqsubseteq \forall S.L_2$, which is discussed in Section 4.1.2. Finally, with guards in place, an instance checking problem can be mapped to a subsumption checking problem, as shown in Section 4.1.3.

4.1.1 Computing $\mathcal{T}_{\mathcal{K}}^{0}$

In this section we convert an $\mathcal{SHIQ}(\mathbb{D})$ knowledge base $\mathcal{K}$ to a TBox by representing individuals by nominals (i.e., in a *controlled* fragment of $\mathcal{SHOIQ}(\mathbb{D})$). Though not necessary, a $\mathcal{SHOIQ}(\mathbb{D})$ concept C is alternatively defined as follows to ease the presentation of the optimization proposed in this section:

$$\begin{array}{lcl} C & ::= & C_d \mid C \sqcap C \mid C \sqcup C \mid \{a\} \mid \neg\{a\} \mid \exists^{\leq n} S.C \mid \exists^{\geq n} S.C \\ C_d & ::= & C_b \mid f < g \mid f = k \\ C_b & ::= & L \mid \top \\ L & ::= & A \mid \neg A \end{array}$$

where k is a finite string. Recall that a complex role S may occur only in concept descriptions of the form $\exists^{\leq 0} S.C_1$ or of the form $\exists^{\geq 1} S.C_1$ (see Section 2.2).

We assume w.l.o.g. that all axioms are in the form of $C_1 \sqsubseteq C_2$. To make this approach fully functional, it is necessary (without loss of generality) that $\mathcal{K}$ only uses qualified *at-most* number restrictions of the form $L_1 \sqsubseteq \exists^{\leq n} R.L_2$, where L_1 and L_2 are atomic concepts or their negations (i.e., literals L). For any $\mathcal{SHIQ}(\mathbb{D})$ knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, Definition 24 establishes a normalized version of $\mathcal{T}$, i.e., $\mathcal{T}^{\text{norm}}$:

Definition 24. ***Normalized*** $\mathcal{SHIQ}(\mathbb{D})$ ***Terminologies***. A $\mathcal{SHIQ}(\mathbb{D})$ constraint $\mathcal{C}$ is *normalized* if it has one of the forms $C_b \sqsubseteq \exists^{\leq n} S.C_b$, $C_L \sqsubseteq C_R$, $S_1 \sqsubseteq S_2$, or $\texttt{Trans}(S)$, where C_L and C_R are defined by the following grammar:

$$\begin{array}{lcl} C_L & ::= & C_d \mid C_L \sqcap C_L \mid C_L \sqcup C_L \mid \exists^{\leq n} S.C_L \\ C_R & ::= & C_d \mid C_R \sqcap C_R \mid C_R \sqcup C_R \mid \exists^{\geq n} S.C_R \end{array}$$

A $\mathcal{SHIQ}(\mathbb{D})$ terminology $\mathcal{T}$ is *normalized* if each constraint $\mathcal{C}$ in $\mathcal{T}$ is normalized. □

It is a straightforward process to obtain an equisatisfiable normalized terminology from an arbitrary $\mathcal{SHIQ}(\mathbb{D})$ terminology $\mathcal{T}$. In particular, we write $\mathcal{T}^{\text{norm}}$ to denote such a terminology, $\bigcup_{\mathcal{C} \in \mathcal{T}} \mathcal{C}^{\text{norm}}$, where $\mathcal{C}^{\text{norm}}$ is obtained by an exhaustive top-to-bottom application of the following rules:

$$
\begin{aligned}
(C_b \sqsubseteq \exists^{\leq n} S.C_b)^{\text{norm}} &= \{C_b \sqsubseteq \exists^{\leq n} S.C_b\} \\
(C_L \sqsubseteq C_R)^{\text{norm}} &= \{C_L \sqsubseteq C_R\} \\
(S_1 \sqsubseteq S_2)^{\text{norm}} &= \{S_1 \sqsubseteq S_2\} \\
(\texttt{Trans}(S))^{\text{norm}} &= \{\texttt{Trans}(S)\} \\
(C_b \sqsubseteq C_1 \sqcap C_2)^{\text{norm}} &= (C_b \sqsubseteq A' \sqcap C_1)^{\text{norm}} \cup (A' \sqsubseteq C_2)^{\text{norm}} \\
(C_b \sqsubseteq C_1 \sqcup C_2)^{\text{norm}} &= (C_b \sqsubseteq A' \sqcup C_1)^{\text{norm}} \cup (A' \sqsubseteq C_2)^{\text{norm}} \\
(C_b \sqsubseteq \exists^{\leq n} S.C)^{\text{norm}} &= \{C_b \sqsubseteq \exists^{\leq n} S.\neg A'\} \cup (A' \sqsubseteq \text{NNF}(\neg C))^{\text{norm}} \\
(C_b \sqsubseteq \exists^{\geq n} S.C)^{\text{norm}} &= \{C_b \sqsubseteq \exists^{\geq n} S.A'\} \cup (A' \sqsubseteq C)^{\text{norm}} \\
(C_1 \sqsubseteq C_2)^{\text{norm}} &= (\neg A' \sqsubseteq \text{NNF}(\neg C_1))^{\text{norm}} \cup (A' \sqsubseteq \text{NNF}(C_2))^{\text{norm}}
\end{aligned}
$$

Note that A' is always a fresh atomic concept and NNF(C) denotes concept C in *negation normal form*, i.e., negations only occurs before atomic concepts. The normalization procedure is linearly bounded by the size of $\mathcal{T}$.

Lemma 4.1.1. *Let $\mathcal{T}$ be an arbitrary $\mathcal{SHIQ}(\mathbb{D})$ terminology. Then:* (1) *If $\mathcal{I} \models \mathcal{T}^{norm}$ for some $\mathcal{I}$, then $\mathcal{I} \models \mathcal{T}$; and* (2) *If $\mathcal{I} \models \mathcal{T}$ for some $\mathcal{I}$, then there is some interpretation $\mathcal{I}'$ over the same domain such that $\mathcal{I}$ and $\mathcal{I}'$ agree on the interpretation of all symbols in $\mathcal{T}$ and $\mathcal{I}' \models \mathcal{T}^{norm}$.*

Proof. (1) The proof is by induction on the normalization rules. The claim holds vacuously for the first four rules. For the fifth rule, because $\mathcal{I} \models C_b \sqsubseteq A' \sqcap C_1$ and $\mathcal{I} \models A' \sqsubseteq C_2$ by the induction hypotheses, it then follows that $\mathcal{I} \models C_b \sqsubseteq C_1 \sqcap C_2$. The sixth rule can be proven analogously. For the seventh rule, we have $\mathcal{I} \models A' \sqsubseteq \text{NNF}(\neg C)$ by the induction hypotheses. Assume $\mathcal{I} \models C_b \sqsubseteq \exists^{\geq n+1} S.C$, then it follows that $\mathcal{I} \models C_b \sqsubseteq \exists^{\geq n+1} S.\neg A'$, which contradicts the given fact that $\mathcal{I} \models C_b \sqsubseteq \exists^{\leq n} S.\neg A'$. The eighth rule can be proven analogously. The last is trivial by observing that $\mathcal{I} \models \neg A' \sqsubseteq \text{NNF}(\neg C_1)$ holds by induction hypotheses, which means $\mathcal{I} \models C_1 \sqsubseteq A'$.

(2) The proof is, again, by induction on the normalization rules and it holds vacuously for the first four rules. For the fifth and sixth rules, we can extend $\mathcal{I}$ to $\mathcal{I}'$ by setting $(A')^{\mathcal{I}'} = (C_2)^{\mathcal{I}'}$. Similarly, for the seventh and eighth rules we set $(\neg A')^{\mathcal{I}'} = (C)^{\mathcal{I}'}$ and $(A')^{\mathcal{I}'} = (C)^{\mathcal{I}'}$, respectively. For the last rule, we define $(A')^{\mathcal{I}'} = (C_2)^{\mathcal{I}'}$. $\square$

Once the TBox of a $\mathcal{SHIQ}(\mathbb{D})$ knowledge base $\mathcal{K}$ is normalized, it is *never* modified in the subsequent processes that compute $\mathcal{T}^i_{\mathcal{K}}, i \geq 0$. This is in contrast to the ABox of $\mathcal{K}$, which is replaced by a TBox as in Definition 25.

Definition 25. ***ABox Conversion***. Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be a $\mathcal{SHIQ}(\mathbb{D})$ knowledge base. A TBox $\mathcal{T}^{\mathcal{A}}$ is defined for $\mathcal{A}$:

$$\begin{aligned}
\mathcal{T}^{\mathcal{A}} = &\ \{\{a\} \sqcap G \sqsubseteq A \mid a : A \in \mathcal{A}\} \\
&\cup \{\{a\} \sqcap G_f \sqsubseteq (f \ \ \mathtt{op} \ \ k) \mid a : (f \ \ \mathtt{op} \ \ k) \in \mathcal{A}\} \\
&\cup \mathcal{T}^{\mathcal{A}}_G \cup \ \mathcal{T}^{\mathcal{A}}_L, \\
\mathcal{T}^{\mathcal{A}}_G = &\ \{\{a\} \sqcap G \sqsubseteq \exists S.\top, \{b\} \sqcap G \sqsubseteq \exists S^-.\top \mid S(a, b) \in \mathcal{A}\}, \\
\mathcal{T}^{\mathcal{A}}_L = &\ \{\{a\} \sqcap G_S \sqsubseteq \exists S.(\{b\} \sqcap G), \{b\} \sqcap G_{S^-} \sqsubseteq \exists S^-.(\{a\} \sqcap G) \mid S(a, b) \in \mathcal{A}\}.
\end{aligned}$$

□

Note that all the axioms resulting from ABox assertions are *guarded* by auxiliary atomic concepts of the form G, G_S, and G_f. Intuitively, these concepts, when coupled with an appropriate absorption, allow a reasoner to *ignore* parts of the original ABox: all the nominals for which G is not *set*. Similarly, for any instance, a reasoner examines only the relevant concrete domain concepts that have the guard G_f set and explores only the relevant instances that have the guard G_S or G_{S^-} set.

In Definition 25, the set of axioms $\mathcal{T}^{\mathcal{A}}_G$ is required to deal with *global typing constraints* in the form of $\top \sqsubseteq \forall S.C$, which are also considered to be *domain* and *range* constraints. Because a global typing constraint of the form $\top \sqsubseteq \forall S.C$ can be absorbed into the axiom $\exists S^-.\top \sqsubseteq C$, the axioms in $\mathcal{T}^{\mathcal{A}}_G$ effectively enable the domain and range constraints to be deposited for the corresponding role assertions.

The separation of $\mathcal{T}^{\mathcal{A}}_L$ from $\mathcal{T}^{\mathcal{A}}$ is for presenting further optimization on universal restrictions introduced in Section 4.1.2. Now, the original ABox has been converted into a TBox $\mathcal{T}^{\mathcal{A}}$, in which guards are used to guide a reasoning algorithm. The normalized TBox $\mathcal{T}^{\text{norm}}$ can then be used to establish how guards can be introduced. In this process, a set of extra axioms may be generated, as shown in Definition 26.

Definition 26. ***TBox Augmentation***. Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be a knowledge base. A TBox $\mathcal{T}^{\mathcal{T}}$ is defined for the TBox of $\mathcal{K}$:

$$\begin{aligned}
\mathcal{T}^{\mathcal{T}} = &\{\ L_1 \sqsubseteq G_S, L_2 \sqsubseteq G_{S^-} \mid L_1 \sqsubseteq \exists^{\leq_n} S.L_2 \in \mathcal{T}^{\text{norm}}, n \geq 0\} \\
&\cup \ \{(t_1 \ \ \mathtt{op} \ \ t_2) \sqsubseteq G_f \mid f \text{ appears in } t_1 \text{ or in } t_2, \text{ and } (t_1 \ \ \mathtt{op} \ \ t_2) \text{ in } \mathcal{T}^{\text{norm}}\} \\
&\cup \ \{G_{S_2} \sqsubseteq G_{S_1}, G_{S_2^-} \sqsubseteq G_{S_1^-} \mid S_1 \sqsubseteq S_2\} \\
&\cup \ \{\top \sqsubseteq G_S \sqcap G_{S^-} \mid S \text{ appears in } \mathcal{T}^{\text{norm}} \text{ and } S \text{ is complex}\}
\end{aligned}$$

$\square$

Observe that, for $L_1 \sqsubseteq \forall S.L_2 \in \mathcal{T}$, the following two axioms are introduced due to $\forall S.L_2 \equiv \exists^{\leq 0} S.\neg L_2$: $L_1 \sqsubseteq G_S, \neg L_2 \sqsubseteq G_{S^-}$. $\mathcal{T}^0_{\mathcal{K}}$ can now be given as follows:

$$\mathcal{T}^0_{\mathcal{K}} = \mathcal{T}^{\text{norm}} \cup \mathcal{T}^{\mathcal{T}} \cup \mathcal{T}^{\mathcal{A}}.$$

An example now follows to illustrate how ABox conversion and TBox augmentation work together to obtain $\mathcal{T}^0_{\mathcal{K}}$.

Example 6. ***Computing*** $\mathcal{T}^0_{\mathcal{K}}$ Consider a knowledge base $\text{univ}_{\mathcal{K}} = (\text{univ}_{\mathcal{T}}, \text{univ}_{\mathcal{A}})$ defined as follows:

$$\begin{aligned}\text{univ}_{\mathcal{T}} &= \{\ \mathit{Dept} \sqsubseteq \forall \mathit{headOf}^-.\mathit{Prof}, \mathit{Chair} \sqsubseteq \mathit{Prof}\ \} \\ \text{univ}_{\mathcal{A}} &= \{\ p : \mathit{Chair}, \mathit{headOf}^-(\mathit{dept}, p)\ \}.\end{aligned}$$

It is easy to see that $(\text{univ}_{\mathcal{T}})^{\text{norm}} = \text{univ}_{\mathcal{T}}$ follows from Definition 24. Also, by Definitions 25 and 26, we obtain the following terminologies:

$$\begin{aligned}\mathcal{T}^{\text{univ}_{\mathcal{A}}} &= \{\{p\} \sqcap G \sqsubseteq \mathit{Chair}, \{\mathit{dept}\} \sqcap G \sqsubseteq \exists \mathit{headOf}^-.\top, \{p\} \sqcap G \sqsubseteq \exists \mathit{headOf}.\top\} \\ &\quad \cup\ \mathcal{T}^{\text{univ}_{\mathcal{A}}}_S \\ \mathcal{T}^{\text{univ}_{\mathcal{A}}}_S &= \{\{\mathit{dept}\} \sqcap G_{\mathit{headOf}^-} \sqsubseteq \exists \mathit{headOf}^-.(\{p\} \sqcap G), \\ &\quad\ \{p\} \sqcap G_{\mathit{headOf}} \sqsubseteq \exists \mathit{headOf}.(\{\mathit{dept}\} \sqcap G)\} \\ \mathcal{T}^{\text{univ}_{\mathcal{T}}} &= \{\mathit{Dept} \sqsubseteq G_{\mathit{headOf}^-}, \neg \mathit{Prof} \sqsubseteq G_{\mathit{headOf}}\} \\ \mathcal{T}^0_{\text{univ}_{\mathcal{K}}} &= \text{univ}_{\mathcal{T}} \cup \mathcal{T}^{\text{univ}_{\mathcal{T}}} \cup \mathcal{T}^{\text{univ}_{\mathcal{A}}}\end{aligned}$$

$\square$

4.1.2 Optimizing Local Universal Restrictions

This section shows how *typing constraints* that are also *local universal restrictions* of the form $L_1 \sqsubseteq \forall S.L_2$ can be leveraged to further optimize ABox absorption [Wu et al., 2013], which differs from the way that global typing constraints are dealt with in Definition 25. In Definition 25, the set $\mathcal{T}^{\mathcal{A}}_L$ is introduced for each assertion $S(a, b)$ occurring in $\mathcal{A}$, and consists of the following binary absorptions:

$$\{a\} \sqcap G_S \sqsubseteq \exists S.(\{b\} \sqcap G) \text{ and } \{b\} \sqcap G_{S^-} \sqsubseteq \exists S^-.(\{a\} \sqcap G).$$

Intuitively, a tableau algorithm starts by generating the successor, the nominal on the right-hand side, after lazy unfolding. Other axioms are subsequently unfolded since the newly introduced nominal includes a guard, for example, the guard G for nominal $\{b\}$. We show how, under some circumstances, one can exploit local universal restrictions to eliminate guards for nominals on the right-hand side of such axioms, possibly replacing the above axioms with the pair

$$\{a\} \sqcap G_S \sqsubseteq \exists S.\{b\} \quad \text{and} \quad \{b\} \sqcap G_{S^-} \sqsubseteq \exists S^-.\{a\},$$

and thereby avoiding subsequent unfolding. Again, Figure 4.1 illustrates this process. In particular: $\mathcal{T}_\mathcal{K}^1$ attempts such eliminations with simple syntactic checks and $\mathcal{T}_\mathcal{K}^2$ uses $\mathcal{T}_\mathcal{K}^1$ for more general subsumption checks to do the same. Details now follow.

Computing $\mathcal{T}_\mathcal{K}^1$

Although computing $\mathcal{T}_\mathcal{K}^1$ requires syntactic checks in the original ABox assertions, it is not necessary to keep the ABox $\mathcal{A}$ after computing $\mathcal{T}_\mathcal{K}^0$. Considering Figure 4.1, when computing $\mathcal{T}_\mathcal{K}^1$, the original ABox $\mathcal{A}$ has already been "discarded". However, since there is a one-to-one correspondence between the original ABox $\mathcal{A}$ and $\mathcal{T}^\mathcal{A}$ (a part of $\mathcal{T}_\mathcal{K}^0$), probing original ABox assertions can be achieved by probing appropriate axioms in $\mathcal{T}^\mathcal{A}$ instead, as given by Definition 25. In particular, for a concept assertion of the form $a : A$ and a role assertion of the form $S(a, b)$, the following holds:

$$a : A \in \mathcal{A} \text{ iff } (\{a\} \sqcap G \sqsubseteq A) \in \mathcal{T}^\mathcal{A}, \tag{4.1}$$

$$S(a, b) \in \mathcal{A} \text{ iff } \{\{a\} \sqcap G_S \sqsubseteq \exists S.(\{b\} \sqcap G), \{b\} \sqcap G_{S^-} \sqsubseteq \exists S^-.(\{a\} \sqcap G)\} \subseteq \mathcal{T}^\mathcal{A}. \tag{4.2}$$

For clarity of presentation, a function, dubbed $\textsc{Map}(\cdot)$, is introduced that maps a concept or role assertion into the set of TBox axioms given by Definition 25. Therefore, (4.1) and (4.2) are equivalent to the following:

$$a : A \in \mathcal{A} \text{ iff } \textsc{Map}(a : A) \subseteq \mathcal{T}^\mathcal{A},$$
$$S(a, b) \in \mathcal{A} \text{ iff } \textsc{Map}(S(a, b)) \subseteq \mathcal{T}^\mathcal{A}.$$

$\mathcal{T}_\mathcal{K}^1$ is given as follows:

$$(\mathcal{T}_\mathcal{K}^0 \backslash \mathcal{T}_L^\mathcal{A}) \ \cup \ \mathcal{T}_\rightarrow^\forall \ \cup \ \mathcal{T}_{\rightarrow d}^\forall \ \cup \ \mathcal{T}_\leftarrow^\forall \ \cup \ \mathcal{T}_{\leftarrow d}^\forall,$$

where each of the terminologies is defined as follows:

$$\begin{aligned} \mathcal{T}_\rightarrow^\forall = \{\{a\} \sqcap G_S \sqsubseteq \exists S.\{b\} \mid\ & \textsc{Map}(S(a, b)) \in \mathcal{T}^\mathcal{A}, \texttt{Trans}(S) \notin \mathcal{T}^{\text{norm}}, \text{ and} \\ & \text{for each } L_1 \sqsubseteq \exists^{\leq n} S.L_2 \in \mathcal{T}^{\text{norm}}, n = 0 \text{ and} \\ & (\textsc{Map}(a : L_1) \cup \textsc{Map}(b : \text{NNF}(\neg L_2))) \cap \mathcal{T}^\mathcal{A} \neq \emptyset\} \end{aligned}$$

$$\begin{aligned}\mathcal{T}^{\forall}_{\rightarrow d} = \{\{a\} \sqcap G_S \sqsubseteq \exists S.(\{b\} \sqcap G) \mid \textsc{Map}(S(a,b)) \in \mathcal{T}^{\mathcal{A}}, \text{and} \\ \text{for some } L_1 \sqsubseteq \exists^{\leq n} S.L_2 \in \mathcal{T}^{\text{norm}}, n > 0 \text{ or} \\ (\textsc{Map}(a : L_1) \cup \textsc{Map}(b : \text{NNF}(\neg L_2))) \cap \mathcal{T}^{\mathcal{A}} = \emptyset\} \\ \mathcal{T}^{\forall}_{\leftarrow} = \{\{b\} \sqcap G_{S^-} \sqsubseteq \exists S^-.\{a\} \mid \textsc{Map}(S(a,b)) \in \mathcal{T}^{\mathcal{A}}, \texttt{Trans}(S) \notin \mathcal{T}^{\text{norm}}, \text{and} \\ \text{for each } L_1 \sqsubseteq \exists^{\leq n} S.L_2 \in \mathcal{T}^{\text{norm}}, n = 0 \text{ and} \\ (\textsc{Map}(a : \text{NNF}(\neg L_1)) \cup \textsc{Map}(b : L_2)) \cap \mathcal{T}^{\mathcal{A}} \neq \emptyset\} \\ \mathcal{T}^{\forall}_{\leftarrow d} = \{\{b\} \sqcap G_{S^-} \sqsubseteq \exists S^-.(\{a\} \sqcap G) \mid \textsc{Map}(S(a,b)) \in \mathcal{T}^{\mathcal{A}}, \text{and} \\ \text{for some } L_1 \sqsubseteq \exists^{\leq n} S.L_2 \in \mathcal{T}^{\text{norm}}, n > 0 \text{ or} \\ (\textsc{Map}(a : \text{NNF}(\neg L_1)) \cup \textsc{Map}(b : L_2)) \cap \mathcal{T}^{\mathcal{A}} = \emptyset\}\end{aligned}$$

Intuitively, $\mathcal{T}^1_{\mathcal{K}}$ includes all components computed in $\mathcal{T}^0_{\mathcal{K}}$, except $\mathcal{T}^{\mathcal{A}}_L$. Because $\mathcal{T}^{\mathcal{A}}_L$ consists of axioms obtained by ABox conversion of role assertions, while such axioms are exactly the ones to be optimized, i.e., it is possible for some guards to be removed. The axioms in $\mathcal{T}^{\mathcal{A}}_L$ can be categorized into two classes: one contains axioms obtained from the ABox conversion step by looking at a role assertion, say, $S(a, b)$, in a *forward* manner, the other in a *backward* manner for the same role assertion, i.e., $S^-(b, a)$. The first perspective enables syntactic checks to remove the guard for the nominal $\{b\}$. If so, a guard-free version (w.r.t. $\{b\}$) of the corresponding axiom in $\mathcal{T}^{\mathcal{A}}_L$ is added to $\mathcal{T}^{\forall}_{\rightarrow}$, otherwise, the original axiom in $\mathcal{T}^{\mathcal{A}}_L$ is retained in $\mathcal{T}^{\forall}_{\rightarrow d}$. The same idea applies to the backward perspective, which generates two sets $\mathcal{T}^{\forall}_{\leftarrow}$ and $\mathcal{T}^{\forall}_{\leftarrow d}$.

The syntactic checks in computing $\mathcal{T}^1_{\mathcal{K}}$ indeed take advantage of the initial assumption of $\mathcal{K}$ consistency. For example, the syntactic checks in computing $\mathcal{T}^{\forall}_{\rightarrow}$ examines if $(\textsc{Map}(a : L_1) \cup \textsc{Map}(b : \text{NNF}(\neg L_2))) \cap \mathcal{T}^{\mathcal{A}} \neq \emptyset$, instead of the more straightforward $\textsc{Map}(b : \text{NNF}(\neg L_2)) \cap \mathcal{T}^{\mathcal{A}} \neq \emptyset$. In the former condition, if $(a : L_1)$ has syntactically occurred in a consistent $\mathcal{K}$, $(b : \text{NNF}(\neg L_2))$ must hold because of the constraint $\exists^{\leq 0} S.L_2$, hence, there is no need to check if $b : \text{NNF}(L_2)$ syntactically appears in $\mathcal{K}$. Therefore, the first condition is more general than the second and is more likely to be satisfied by a consistent $\mathcal{K}$.

Computing $\mathcal{T}^2_{\mathcal{K}}$

Recall that no reasoning is required in computing $\mathcal{T}^1_{\mathcal{K}}$. Instead, syntactic checks for concept assertions of the form $a : L_1$ or $b : L_2$ are performed through a mapping function over $\mathcal{T}^0_{\mathcal{K}}$. If these concept assertions are found, then it is guaranteed that $S(a, b)$, together with the axioms resulting from these concept assertions, is consistent with any local universal restrictions of the form $L_1 \sqsubseteq \forall S.L_2$. Although such checks are far from complete, $\mathcal{T}^1_{\mathcal{K}}$ can now be used to perform subsumption checks to find additional cases where local universal restrictions are satisfied by role assertions, that is, $\mathcal{T}^1_{\mathcal{K}}$ can be used to compute $\mathcal{T}^2_{\mathcal{K}}$.

The purpose of computing $\mathcal{T}^2_{\mathcal{K}}$ is then simple: those axioms in $\mathcal{T}^{\forall}_{\rightarrow d}$ or $\mathcal{T}^{\forall}_{\leftarrow d}$ obtained in computing $\mathcal{T}^1_{\mathcal{K}}$, i.e., ones that do not meet the syntactic checking conditions for removing guards, are further examined. The difference is that $\mathcal{T}^2_{\mathcal{K}}$ will employ semantic checks instead of syntactic checks. Hence, $\mathcal{T}^2_{\mathcal{K}}$ is given by $(\mathcal{T}^1_{\mathcal{K}} \backslash \mathcal{T}^{sub}) \cup \mathcal{T}^{add}$, where $\mathcal{T}^{add}$ and $\mathcal{T}^{sub}$ are defined as follows:

$$
\begin{aligned}
\mathcal{T}^{sub} = \{\ & \{a\} \sqcap G_S \sqsubseteq \exists S.(\{b\} \sqcap G) \mid \mathtt{Trans}(S) \notin \mathcal{T}^{\text{norm}}, \\
& \quad \text{for each } L_1 \sqsubseteq \exists^{\leq n} S.L_2 \in \mathcal{T}^{\text{norm}} : n = 0 \text{ and} \\
& \quad\quad (\mathcal{T}^1_{\mathcal{K}} \models \{a\} \sqcap G \sqsubseteq L_1 \text{ or } \mathcal{T}^1_{\mathcal{K}} \models \{b\} \sqcap G \sqsubseteq \neg L_2), \\
& \quad \text{and } \{a\} \sqcap G_S \sqsubseteq \exists S.(\{b\} \sqcap G) \in \mathcal{T}^{\forall}_{\rightarrow d}\} \\
\cup\ \{\ & \{b\} \sqcap G_{S^-} \sqsubseteq \exists S^-.(\{a\} \sqcap G) \mid \mathtt{Trans}(S^-) \notin \mathcal{T}^{\text{norm}}, \\
& \quad \text{for each } L_1 \sqsubseteq \exists^{\leq n} S.L_2 \in \mathcal{T}^{\text{norm}} : n = 0 \text{ and} \\
& \quad\quad (\mathcal{T}^1_{\mathcal{K}} \models \{a\} \sqcap G \sqsubseteq \neg L_1 \text{ or } \mathcal{T}^1_{\mathcal{K}} \models \{b\} \sqcap G \sqsubseteq L_2), \\
& \quad \text{and } \{b\} \sqcap G_{S^-} \sqsubseteq \exists S^-.(\{a\} \sqcap G) \in \mathcal{T}^{\forall}_{\leftarrow d}\ \} \\
\mathcal{T}^{add} = \{ & \{a\} \sqcap G_S \sqsubseteq \exists S.\{b\} \mid \{a\} \sqcap G_S \sqsubseteq \exists S.(\{b\} \sqcap G) \in \mathcal{T}^{sub}\}
\end{aligned}
$$

Intuitively, $\mathcal{T}^{sub}$ collects all the axioms that pass the semantic checks and $\mathcal{T}^{add}$ consists of the guard-free version of the axioms in $\mathcal{T}^{sub}$.

Example 7. ***Computing*** $\mathcal{T}^1_{\mathcal{K}}$ ***and*** $\mathcal{T}^2_{\mathcal{K}}$ Consider the knowledge base $\text{univ}_{\mathcal{K}}$ defined in Example 6 and $\mathcal{T}^0_{\text{univ}_{\mathcal{K}}}$ computed for $\text{univ}_{\mathcal{K}}$. To compute $\mathcal{T}^1_{\mathcal{K}}$, the syntactic check $(\textsc{Map}(\mathit{dept} : \mathit{Dept}) \cup \textsc{Map}(p : \mathit{Prof})) \subseteq \mathcal{T}^{\text{univ}_{\mathcal{A}}}$ is performed for the forward direction and the syntactic check $(\textsc{Map}(p : \neg\mathit{Prof}) \cup \textsc{Map}(\mathit{dept} : \neg\mathit{Dept})) \subseteq \mathcal{T}^{\text{univ}_{\mathcal{A}}}$ for the backward direction. However, neither of the checks succeed. Thus, $\mathcal{T}^1_{\mathcal{K}}$ is computed as follows for this knowledge base:

$$
\begin{aligned}
\mathcal{T}^{\forall}_{\rightarrow} &= \mathcal{T}^{\forall}_{\leftarrow} = \emptyset \\
\mathcal{T}^{\forall}_{\rightarrow d} &= \{\{\mathit{dept}\} \sqcap G_{\mathit{headOf}^-} \sqsubseteq \exists \mathit{headOf}^-.(\{p\} \sqcap G)\} \\
\mathcal{T}^{\forall}_{\leftarrow d} &= \{\{p\} \sqcap G_{\mathit{headOf}} \sqsubseteq \exists \mathit{headOf}.(\{\mathit{dept}\} \sqcap G)\} \\
\mathcal{T}^1_{\text{univ}_{\mathcal{K}}} &= (\mathcal{T}^0_{\text{univ}_{\mathcal{K}}} \backslash \mathcal{T}^{\text{univ}_{\mathcal{A}}}_S) \cup \mathcal{T}^{\forall}_{\rightarrow d} \cup \mathcal{T}^{\forall}_{\leftarrow d}
\end{aligned}
$$

$\mathcal{T}^1_{\text{univ}_{\mathcal{K}}}$ in this particular case is not effective for removing guards, but it is used to compute $\mathcal{T}^2_{\text{univ}_{\mathcal{K}}}$ by checking if the typing constraint $\mathit{Dept} \sqsubseteq \forall \mathit{headOf}^-.\mathit{Prof}$ is satisfied by the role assertion $headOf^{-}(dept, p)$. Since $\mathcal{T}^1_{\text{univ}_\mathcal{K}} \models \{p\} \sqcap G \sqsubseteq Prof$, $\mathcal{T}^2_{\text{univ}_\mathcal{K}}$ can be obtained:

$$\begin{aligned}
\mathcal{T}^{sub} &= \{\{dept\} \sqcap G_{headOf^-} \sqsubseteq \exists headOf^-.(\{p\} \sqcap G)\} \\
\mathcal{T}^{add} &= \{\{dept\} \sqcap G_{headOf^-} \sqsubseteq \exists headOf^-.\{p\}\} \\
\mathcal{T}^2_{\text{univ}_\mathcal{K}} &= (\mathcal{T}^1_{\text{univ}_\mathcal{K}} \backslash \mathcal{T}^{sub}) \ \cup \ \mathcal{T}^{add}
\end{aligned}$$

□

4.1.3 Instance Checking as Subsumption Checking

The main results are given in this section, which shows that an instance checking problem over a $\mathcal{SHIQ}(\mathbb{D})$ knowledge base $\mathcal{K}$ can be mapped to a subsumption checking problem over the $\mathcal{SHOIQ}(\mathbb{D})$ TBox $\mathcal{T}^i_\mathcal{K}$, for $0 \leq i \leq 2$. Such subsumption checks require the notion of a *derivative concept*, which serve the purpose of introducing guards appropriately for any query concept. The formal definition follows:

Definition 27. *Derivative Concept*. The *derivative concept* $\mathfrak{D}_C$ of a general $\mathcal{SHIQ}(\mathbb{D})$ concept C is defined as follows:

$$\mathfrak{D}_C = \begin{cases} \top & \text{if } C = C_b; \\ \bigsqcap G_{f_i} & \text{if } C = (t_1 \ \ \mathtt{op} \ \ t_2) \text{ and } f_i \text{ appears in } t_1 \text{ or } t_2; \\ \mathfrak{D}_{C_1} \sqcap \mathfrak{D}_{C_2} & \text{if } C = C_1 \sqcap C_2 \text{ or } C = C_1 \sqcup C_2; \\ G_S \sqcap \forall S.(\mathfrak{D}_{C_1} \sqcap G) & \text{if } C = \exists^{\geq n} S.C_1 \text{ or } C = \exists^{\leq n} S.C_1. \end{cases}$$

□

Theorem 4.1.2. *For any consistent $\mathcal{SHIQ}(\mathbb{D})$ knowledge base $\mathcal{K}$, concept C, individual a, and $0 \leq i \leq 2$:*

$$\mathcal{K} \models a : C \quad \textit{iff} \quad \mathcal{T}^i_\mathcal{K} \models \{a\} \sqcap \mathrm{D} \sqsubseteq C, \ \textit{where } \mathrm{D} = G \sqcap \mathfrak{D}_C.$$

Proof. Cases $i = 0$ and $i = 1$ are implicitly included in the case $i = 2$, so, it is sufficient to prove the latter case. The *only-if* direction is equivalent to the following claim: if $\mathcal{T}^2_\mathcal{K} \not\models \{a\} \sqcap \mathrm{D} \sqsubseteq \mathrm{C}$, then $\mathcal{K} \not\models a : \mathrm{C}$. Assume that there is an interpretation $\mathcal{I}_0$ that satisfies $\mathcal{T}^2_\mathcal{K}$ such that $(\{a\})^{\mathcal{I}_0} \subseteq (\mathrm{D})^{\mathcal{I}_0}$ but $(\{a\})^{\mathcal{I}_0} \cap (\mathrm{C})^{\mathcal{I}_0} = \emptyset$ and an interpretation $\mathcal{I}_1$ that satisfies $\mathcal{K}$ in which *all at-least restrictions are fulfilled by anonymous objects*. Hence, we do not need to consider at-least restrictions, no matter how expressed, in the construction below. Without loss of generality, we assume both $\mathcal{I}_0$ and $\mathcal{I}_1$ are tree-shaped outside of the ABox (converted ABox). Our proof proceeds by building an interpretation $\mathcal{J}$ such that $\mathcal{J}$ satisfies $\mathcal{K}$ and $(a)^{\mathcal{J}} \notin (C)^{J}$.

The construction of the interpretation $\mathcal{J}$ for $\mathcal{K} \cup \{a : \neg \mathrm{C}\}$ follows. Let $\Gamma^{\mathcal{I}_0}$ be the set of objects $o \in \Delta^{\mathcal{I}_0}$ such that either $o \in (\{a\})^{\mathcal{I}_0}$ and $(\{a\})^{\mathcal{I}_0} \subseteq (G)^{\mathcal{I}_0}$ or o is an anonymous object in $\Delta^{\mathcal{I}_0}$ rooted by such an object. Similarly let $\Gamma^{\mathcal{I}_1}$ be the set of objects $o \in \Delta^{\mathcal{I}_1}$ such that either $o \in (\{a\})^{\mathcal{I}_1}$ and $(\{a\})^{\mathcal{I}_0} \cap (G)^{\mathcal{I}_0} = \emptyset$ or o is an anonymous object in $\Delta^{\mathcal{I}_1}$ rooted by such an object. We stipulate the following protocols for constructing $\mathcal{J}$:

1. $\Delta^{\mathcal{J}} = \Gamma^{\mathcal{I}_0} \cup \Gamma^{\mathcal{I}_1}$;
2. $(a)^{\mathcal{J}} \in (\{a\})^{\mathcal{I}_0}$ for $(a)^{\mathcal{J}} \in \Gamma^{\mathcal{I}_0}$ and $(a)^{\mathcal{J}} = (a)^{\mathcal{I}_1}$ for $(a)^{\mathcal{J}} \in \Gamma^{\mathcal{I}_1}$;
3. $o \in A^{\mathcal{J}}$ if $o \in A^{\mathcal{I}_0}$ and $o \in \Gamma^{\mathcal{I}_0}$ or if $o \in A^{\mathcal{I}_1}$ and $o \in \Gamma^{\mathcal{I}_1}$ for an atomic concept A (similarly for concrete domain concepts of the form $(t_1 \ \mathtt{op} \ \ t_2)$);
4. $(o_1, o_2) \in (S)^{\mathcal{J}}$ if
   (a) $(o_1, o_2) \in S^{\mathcal{I}_0}$ and $\{o_1, o_2\} \subseteq \Gamma^{\mathcal{I}_0}$, or $(o_1, o_2) \in S^{\mathcal{I}_1}$ and $\{o_1, o_2\} \subseteq \Gamma^{\mathcal{I}_1}$; or
   (b) $o_1 \in (\{a\})^{\mathcal{I}_0} \cap (G)^{\mathcal{I}_0}$, $o_2 \in (\{b\})^{\mathcal{I}_1}$, and $S(a, b) \in \mathcal{A}$; or
   (c) $o_1 \in (\{a\})^{\mathcal{I}_1}$, $o_2 \in (\{b\})^{\mathcal{I}_0} \cap (G)^{\mathcal{I}_0}$, and $S(a, b) \in \mathcal{A}$; or
   (d) $(o_1, o_2) \in S'^{\mathcal{J}}$ and $S' \sqsubseteq^{*}_{\mathcal{R}} S$; or
   (e) $(o_1, o_3) \in S^{\mathcal{J}}$, $(o_3, o_2) \in S^{\mathcal{J}}$, and $\mathtt{Trans}(S) \in \mathcal{K}$.

The construction of $\mathcal{J}$ is now complete. Before proving the main result, Lemma 4.1.3 establishes a useful property for transitive roles:

Lemma 4.1.3. *For $\{o_1, o_2\} \subseteq \Delta^{\mathcal{J}}$, if $(o_1, o_2) \in (S)^{\mathcal{J}}$ and $\mathtt{Trans}(S) \in \mathcal{K}$, then either $\{o_1, o_2\} \subseteq \Gamma^{\mathcal{I}_0}$ or $\{o_1, o_2\} \subseteq \Gamma^{\mathcal{I}_1}$*

Proof. The proof proceeds by induction on all cases for interpretation of roles (i.e., the 4th point) in defining $\mathcal{J}$. Case 4a is trivial; cases 4b and 4c are not applicable when $\mathtt{Trans}(S) \in \mathcal{K}$ as otherwise by the definition of $\mathcal{T}^{\mathcal{T}}$ it holds that $o_1 \in (G_S)^{\mathcal{I}_0}$ and thus the contradiction $o_2 \in (G)^{\mathcal{I}_0}$. Case 4d is trivial by the induction hypothesis if $\mathtt{Trans}(S_1) \in \mathcal{K}$. Considering $\mathtt{Trans}(S_1) \notin \mathcal{K}$, we show this case is not applicable. Suppose $o_1 \in (\{a\})^{\mathcal{I}_0} \cap (G)^{\mathcal{I}_0}$ and $o_2 \in (\{b\})^{\mathcal{I}_1}$ (or vice versa), then this is only possible through cases 4b or 4c. While a similar contradiction can be drawn as in case 4b or 4c because of $G_S \sqsubseteq G_{S_1}$, i.e., $o_1 \in (G_{S_1})^{\mathcal{I}_0}$ and thus the contradiction $o_2 \in (G)^{\mathcal{I}_0}$. Case 4e follows from the induction hypotheses because either $\{o_1, o_2, o'\} \subseteq \Gamma^{\mathcal{I}_0}$ or $\{o_1, o_2, o'\} \subseteq \Gamma^{\mathcal{I}_1}$ hold. □

Now we show $\mathcal{J}$ satisfies $\mathcal{K}$ and $(a)^{\mathcal{J}} \notin (C)^{\mathcal{J}}$. For the first claim, i.e., $\mathcal{J} \models \mathcal{K}$, it suffices to consider only the S edges crossing the two interpretations. The edges in protocol 4a satisfy all axioms in $\mathcal{K}$ as the remainder of the interpretation $\mathcal{J}$ is copied from one of the two interpretations that satisfy $\mathcal{K}$. Now we consider the two cases as defined in protocols 4b and 4c. Note that none of these edges need to fulfill at-least qualified number restrictions, which are already fulfilled (potentially redundantly) by anonymous objects whenever possible. Furthermore, role hierarchy and transitive roles are satisfied by cases 4d and 4e, respectively. Therefore, only at-most qualified number restrictions (including universal restrictions) need to be considered for 4b – 4e.

For cases 4b and 4c, consider an axiom expressing an *at-most* restriction in the form of $L_1 \sqsubseteq \exists^{\leq n} S.L_2 \in \mathcal{T}$. There are two possibilities: in one case, we can conclude $o_1 \notin (L_1)^{\mathcal{I}_0}$ as otherwise $o_1 \in (G_S)^{\mathcal{I}_0}$ by the definition of $\mathcal{T}^{\mathcal{T}}$ and thus $o_2 \in (G)^{\mathcal{I}_0}$ by the rules for construction of $\mathcal{T}^2_{\mathcal{K}}$, which contradicts our assumption that $(\{b\})^{\mathcal{I}_0} \cap (G)^{\mathcal{I}_0} = \emptyset$, hence this at-most restriction is satisfied vacuously; in the other case, we cannot derive a contradiction because G was removed by our optimization shown in Sect. 4.1.2, then it must be the case that the axiom $L_1 \sqsubseteq \exists^{\leq 0} S.L_2 \in \mathcal{T}$, i.e., $L_1 \sqsubseteq \forall S.\neg L_2$, has been satisfied by the role assertion $S(a, b)$. For 4e, Lemma 4.1.3 stipulates that in case (4d) either $\{o_1, o_2, o'\} \subseteq \Gamma^{\mathcal{I}_0}$ or $\{o_1, o_2, o'\} \subseteq \Gamma^{\mathcal{I}_1}$ hold; hence any universal restriction of the form $L_1 \sqsubseteq \forall S.L_2$ (recall that concepts of the form $\exists^{\leq n} S.L_2$ are disallowed for complex S when $n \neq 0$) must be satisfied by (o_1, o_2) because it is already satisfied by (o_1, o_2) in $\mathcal{I}_0$ ($\mathcal{I}_1$, respectively). Edges from case 4d are trivial extension to all of the above. Hence all inclusion axioms in $\mathcal{K}$ are satisfied by $\mathcal{J}$.

To show $(a)^{\mathcal{J}} \notin (C)^{\mathcal{J}}$, one should note that a is in the initial query, hence, $(a)^{\mathcal{J}} \in \Gamma^{\mathcal{I}_0}$ and $(a)^{\mathcal{J}} \in (G)^{\mathcal{I}_0}$. In addition, we have $(\{a\})^{\mathcal{I}_0} \subseteq (\mathrm{D})^{\mathcal{I}_0}$ and $(\{a\})^{\mathcal{I}_0} \cap (\mathrm{C})^{\mathcal{I}_0} = \emptyset$ as our assumption. We show that $(a)^{\mathcal{J}} \notin (C)^{\mathcal{J}}$ holds using structural induction, considering the query concept C to be in NNF. The induction hypothesis is that, for any a, if $(\{a\})^{\mathcal{I}_0} \subseteq (D')^{\mathcal{I}_0}$ and $(\{a\})^{\mathcal{I}_0} \cap (\mathrm{C}')^{\mathcal{I}_0} = \emptyset$ then $(a)^{\mathcal{J}} \notin (C')^{\mathcal{J}}$, where C' is a subexpression in C and $D' = G \sqcap \mathfrak{D}_{C'}$.

- $C = A$ or $C = (t_1 \;\mathsf{op}\; t_2)$, then $(a)^{\mathcal{J}} \notin (C)^{\mathcal{J}}$ holds trivially by protocol 3: $(a)^{\mathcal{J}} \in \Gamma^{\mathcal{I}_0}$ and $(a)^{\mathcal{J}} \notin C^{\mathcal{I}_0}$ (due to our assumption that $(\{a\})^{\mathcal{I}_0} \cap (\mathrm{C})^{\mathcal{I}_0} = \emptyset$).
- $C = \exists S.C'$. By way of contradiction, we assume $(a)^{\mathcal{J}} \in (\exists S.C')^{\mathcal{J}}$, which means there is an individual b such that $(b)^{\mathcal{J}} \in (C')^{\mathcal{J}}$ and $((a)^{\mathcal{J}}, (b)^{\mathcal{J}}) \in (S)^{\mathcal{J}}$.
    1. If $(b)^{\mathcal{J}} \in \Gamma^{\mathcal{I}_0}$, then $(b)^{\mathcal{J}} \in (\{b\})^{\mathcal{I}_0}, ((\{a\})^{\mathcal{I}_0}, (\{b\})^{\mathcal{J}}) \in (S)^{\mathcal{I}_0}$. By our initial assumption, $(\{a\})^{\mathcal{I}_0} \subseteq (G \sqcap \mathfrak{D}_C)^{\mathcal{I}_0}$, where $\mathfrak{D}_C = G_S \sqcap \forall S.\mathfrak{D}_{C'}$, hence, we have $(\{b\})^{\mathcal{I}_0} \subseteq (\mathfrak{D}_{C'})^{\mathcal{I}_0}$, which, together with $(\{b\})^{\mathcal{I}_0} \subseteq (G)^{\mathcal{I}_0}$, implies that $(\{b\})^{\mathcal{I}_0} \subseteq$

$(D')^{\mathcal{I}_0}$. However, by the induction hypothesis and $(b)^{\mathcal{J}} \in (C')^{\mathcal{J}}$, either $(\{b\})^{\mathcal{I}_0} \not\subseteq (D')^{\mathcal{I}_0}$ or $(\{b\})^{\mathcal{I}_0} \subseteq (\mathrm{C}')^{\mathcal{I}_0}$, so the latter must hold. The latter, nevertheless, contradicts our original assumption that $(\{a\})^{\mathcal{I}_0} \cap (\mathrm{C})^{\mathcal{I}_0} = \emptyset$, with $C = \exists S.C'$.

2. If $(b)^{\mathcal{J}} \in \Gamma^{\mathcal{I}_1}$, then $(\{b\})^{\mathcal{I}_0} \cap (G)^{\mathcal{I}_0} = \emptyset$. By our initial assumption, $(\{a\})^{\mathcal{I}_0} \subseteq (G \sqcap \mathfrak{D}_C)^{\mathcal{I}_0}$, where $\mathfrak{D}_C = G_S \sqcap \forall S.\mathfrak{D}_{C'}$, hence, $(\{a\})^{\mathcal{I}_0} \subseteq (G \sqcap G_S)^{\mathcal{I}_0}$. Since $S(a, b) \in \mathcal{A}$, by Definition 25 it is easy to see that $(\{b\})^{\mathcal{I}_0} \subseteq (G)^{\mathcal{I}_0}$: a contradiction.

- $C = \forall S.C'$. Observe that $\forall S.C'$ is a shorthand for $\exists^{\leq_0} S.\neg C'$, so $\mathfrak{D}_{\forall S.C'} = \mathfrak{D}_{\exists S.C'}$. Assume, by way of contradiction, that $(a)^{\mathcal{J}} \in (\forall S.C')^{\mathcal{J}}$. Because $(\{a\})^{\mathcal{I}_0} \not\subseteq (\forall S.C')^{\mathcal{I}_0}$, there is a nominal $\{b\}$ such that $((\{a\})^{\mathcal{I}_0}, (\{b\})^{\mathcal{I}_0}) \in (S)^{\mathcal{I}_0}$ and $(\{b\})^{\mathcal{I}_0} \not\subseteq (C')^{\mathcal{I}_0}$. Thus, we have $(\{b\}) \in \Gamma^{\mathcal{I}_0}$ by definition and $((\{a\})^{\mathcal{J}}, (\{b\})^{\mathcal{J}}) \in (S)^{\mathcal{J}}$ by protocol 4a. Since $(a)^{\mathcal{J}} \in (\forall S.C')^{\mathcal{J}}$, it must be that $(b)^{\mathcal{J}} \in (C')^{\mathcal{J}}$. However, $(\{a\})^{\mathcal{I}_0} \subseteq (\mathrm{D})^{\mathcal{I}_0}$ implies that $(\{a\})^{\mathcal{I}_0} \subseteq (\forall S.\mathfrak{D}_{C'})^{\mathcal{I}_0}$, which means $(\{b\})^{\mathcal{I}_0} \subseteq (\mathfrak{D}_{C'})^{\mathcal{I}_0}$, hence, by the induction hypothesis, $(b)^{\mathcal{J}} \notin (C')^{\mathcal{J}}$: a contradiction.

- $C = \exists^{\leq_n} S.C'$. The proof follows immediately from the cases $C = \forall S.\neg C'$ if $n = 0$. Otherwise, by assuming $(a)^{\mathcal{J}} \in (\exists^{\leq_n} S.C')^{\mathcal{J}}$, the case analyses are similar to that of the case $C = \exists S.C'$.

- $C = \neg A$. Because $(\{a\})^{\mathcal{I}_0} \cap C^{\mathcal{I}_0} = \emptyset$, we have $(\{a\})^{\mathcal{I}_0} \subseteq (A)^{\mathcal{I}_0}$, which by protocol 3, together with $(a)^{\mathcal{J}} \in (\{a\})^{\mathcal{I}_0}$ and $(a)^{\mathcal{J}} \in \Gamma^{\mathcal{I}_0}$, implies that $(a)^{\mathcal{J}} \in A^{\mathcal{J}}$, i.e., $(a)^{\mathcal{J}} \notin (C)^{\mathcal{J}}$.

- $C = C_1 \sqcap C_2$. Assume, by way of contradiction, that $(a)^{\mathcal{J}} \in (C)^{\mathcal{J}}$, then $(a)^{\mathcal{J}} \in (C_1)^{\mathcal{J}}$ and $(a)^{\mathcal{J}} \in (C_2)^{\mathcal{J}}$. By the induction hypothesis and $(a)^{\mathcal{J}} \in (C_1)^{\mathcal{J}}$, we have either $(\{a\})^{\mathcal{I}_0} \not\subseteq (\mathrm{D}_1)^{\mathcal{I}_0}$ or $(\{a\})^{\mathcal{I}_0} \subseteq (\mathrm{C}_1)^{\mathcal{I}_0}$. By $(\{a\})^{\mathcal{I}_0} \subseteq (G)^{\mathcal{I}_0}, (\{a\})^{\mathcal{I}_0} \subseteq (\mathfrak{D}_C)^{\mathcal{I}_0}$ and $\mathfrak{D}_C \sqsubseteq \mathfrak{D}_{C_1}$, we have $(\{a\})^{\mathcal{I}_0} \subseteq (G \sqcap \mathfrak{D}_{C_1})^{\mathcal{I}_0}$, i.e., $(\{a\})^{\mathcal{I}_0} \subseteq (\mathrm{D}_1)^{\mathcal{I}_0}$, so it must be the case that $(\{a\})^{\mathcal{I}_0} \subseteq (\mathrm{C}_1)^{\mathcal{I}_0}$. For the same reason, $(\{a\})^{\mathcal{I}_0} \subseteq (\mathrm{C}_2)^{\mathcal{I}_0}$. Hence, $(\{a\})^{\mathcal{I}_0} \subseteq (\mathrm{C}_1 \sqcap C_2)^{\mathcal{I}_0}$, which contradicts the initial assumption $(\{a\})^{\mathcal{I}_0} \subseteq (C)^{\mathcal{I}_0}$.

The *if* direction, equivalent to the claim that if $\mathcal{K} \not\models a : \mathrm{C}$ then $\mathcal{T}^2_{\mathcal{K}} \not\models \{a\} \sqcap \mathrm{D} \sqsubseteq \mathrm{C}$, holds by observing that if $\mathcal{K} \cup \{a : \neg\mathrm{C}\}$ is satisfiable then the satisfying interpretation I can be extended to $(G)^I = (G_f)^I = (G_S)^I = \Delta^I$ and $(\{a\})^I = \{a^I\}$ for all individuals a, concrete features f, and roles S. This extended interpretation then satisfies $\mathcal{T}^2_{\mathcal{K}}$ and $(\{a\})^I \subseteq (\mathrm{D})^I \cap (\neg\mathrm{C})^I$.

□

The main idea of the proof of Theorem 4.1.2 is illustrated in Figure 4.2, and it is dubbed *a donut theorem*. The donut theorem ensures that a tinbit (a wordplay of *tiny bit*), which is dynamically generated by guarded reasoning, can be combined with a *donut* derived from an interpretation $\mathcal{I}_1$ of the original knowledge base $\mathcal{K}$ to obtain an interpretation $\mathcal{J}$ satisfying all constraints and assertions in $\mathcal{K}$ and for which a occurs in the interpretation of concept $\neg C$. Although the purpose of the donut theorem is to derive $\mathcal{J}$ from a tinbit and a donut, a tinbit can be used to derive an interpretation for the original $\mathcal{T}^i_{\mathcal{K}}$, i.e., $\mathcal{I}_0$ in Figure 4.2. Note that the assumption of knowledge base consistency and of internal support for binary absorption are crucial in this setting, e.g., to ensure that tinbits for any instance checking problem are indeed tiny bits of (a description of) $\mathcal{J}$.

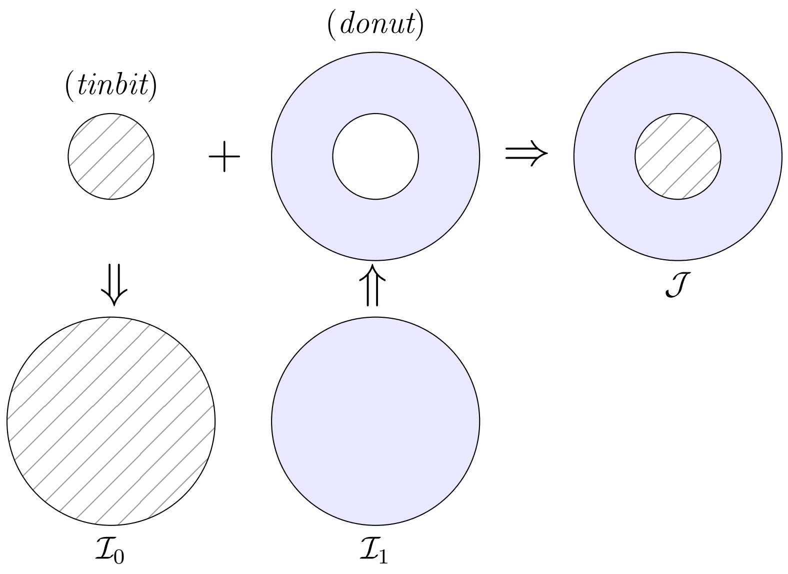


Figure 4.2: A donut theorem: ensuring interpretations.

In $\mathcal{SHIQ}(\mathbb{D})$, which on its own cannot equate nominals, there is no need to rely explicitly on the unique name assumption (see Definition 5). However, explicit equalities and inequalities can be enabled in a ABox and be processed similarly to Definition 25, e.g., $a \approx b$ to $\{a\} \sqcap G \sqsubseteq \{b\} \sqcap G$ and vice versa and so on. This is sufficient for the construction of the interpretation $\mathcal{J}$ in the proof of Theorem 4.1.2 to go through. Note that the interpretations of nominals for which G is not set in $\mathcal{I}_0$ are irrelevant for constructing the interpretation $\mathcal{J}$ even though there could be axioms of the form $\top \sqsubseteq \mathrm{C}$ that are applicable to such constants (one could even augment all such axioms with guards to avoid this effect). Therefore, those nominals can be ignored completely during reasoning and, thus, nodes corresponding to the nominals can be generated *lazily* on demand, driven by the guards G and G_S.

4.2 On Generalized Binary Absorption

A guard G is effective only if it is "observed" simultaneously with a nominal concept. Recall that binary absorption was originally defined for axioms of the form $A_1 \sqcap A_2 \sqsubseteq C$, where $\{A_1, A_2\} \subseteq \mathrm{NC}$. To accommodate nominals introduced in the process presented in the previous section, this section properly extends binary absorption to ensure that guarded nominals are reasoned about only if a guard is seen. Nevertheless, using other absorption algorithms is unable to retain the effects of guards. Consequently, Section 4.2.2 expands on an extension to binary absorption that functions on the converted knowledge base $\mathcal{T}_{\mathcal{K}}$. To define such an extension, the notion of witnesses needs to be introduced prior to absorption, as shown in the next section.

4.2.1 On Witnesses

Recall that tableau algorithms for checking the satisfaction of a concept C operate by manipulating a completion graph (see Section 2.2), which encodes a *partial description* of (eventual) interpretations $\mathcal{I}$ for which $(C)^{\mathcal{I}}$ will be non-empty. Such a graph will almost always abstract details on class membership for hypothetical elements of $\Delta^{\mathcal{I}}$ and on details relating to the interpretation of roles. To talk formally about absorption and lazy evaluation, it is necessary to codify the idea of a completion graph. This has been done in [Horrocks and Tobies, 2000b] by introducing the notion of a *witness*, of an interpretation that *stems* from a witness, and of what it means for a witness to be *admissible* with respect to a given terminology.

Definition 28. ***Witness***. Let C be an $\mathcal{SHOIQ}(\mathbb{D})$ concept.[1] A *witness* $\mathcal{W} = (\Delta^{\mathcal{W}}, \cdot^{\mathcal{W}}, \mathcal{L}^{\mathcal{W}})$ for C consists of a non-empty set $\Delta^{\mathcal{W}}$, a function $\cdot^{\mathcal{W}}$ that maps NR to subsets of $\Delta^{\mathcal{W}} \times \Delta^{\mathcal{W}}$, and a function $\mathcal{L}^{\mathcal{W}}$ that maps $\Delta^{\mathcal{W}}$ to sets of $\mathcal{SHOIQ}(\mathbb{D})$ concepts such that:

(W1) there is some $x \in \Delta^{\mathcal{W}}$ with $C \in \mathcal{L}^{\mathcal{W}}(x)$,

(W2) there is an interpretation $\mathcal{I}$ that *stems* from $\mathcal{W}$, and

(W3) for each $\mathcal{I}$ that *stems* from $\mathcal{W}$, $x \in (C)^{\mathcal{I}}$ if $C \in \mathcal{L}^{\mathcal{W}}(x)$.

[1]The definition of witness can be abstracted for any DLs that have $\mathcal{ALCIO}$ as a sublanguage and that satisfy some criteria on the interpretations stated in [Horrocks and Tobies, 2000b].

An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, (\cdot)^{\mathcal{I}})$ is said to *stem* from $\mathcal{W}$ if $\Delta^{\mathcal{I}} = \Delta^{\mathcal{W}}$, $(\cdot)^{\mathcal{I}}|_{\mathrm{NR}} = \cdot^{\mathcal{W}}$, for each $A \in \mathrm{NC}$, $A \in \mathcal{L}^{\mathcal{W}}(x)$ implies $x \in (A)^{\mathcal{I}}$ and $\neg A \in \mathcal{L}^{\mathcal{W}}(x)$ implies $x \notin (A)^{\mathcal{I}}$, for each $a \in \mathrm{NI}$, $\{a\} \in \mathcal{L}^{\mathcal{W}}(x)$ implies $x \in (\{a\})^{\mathcal{I}}$ and $\neg\{a\} \in \mathcal{L}^{\mathcal{W}}(x)$ implies $x \notin (\{a\})^{\mathcal{I}}$, for each $(f \; \mathsf{op} \; k)$, $(f \; \mathsf{op} \; k) \in \mathcal{L}^{\mathcal{W}}(x)$ implies $x \in ((f \; \mathsf{op} \; k))^{\mathcal{I}}$ and $\neg(f \; \mathsf{op} \; k) \in \mathcal{L}^{\mathcal{W}}(x)$ implies $x \notin ((f \; \mathsf{op} \; k))^{\mathcal{I}}$.

A witness $\mathcal{W}$ is called *admissible* with respect to a TBox $\mathcal{T}$ if there is an interpretation $\mathcal{I}$ that stems from $\mathcal{W}$ with $\mathcal{I} \models \mathcal{T}$. □

The properties satisfied by a witness are presented in the following lemmas, originally shown in [Horrocks and Tobies, 2000b].

Lemma 4.2.1. *Let L be a DL. A concept $C \in L$ is satisfiable w.r.t. a TBox $\mathcal{T}$ iff it has a witness that is admissible w.r.t. $\mathcal{T}$.*

Lemma 4.2.2. *Let L, C, $\mathcal{T}$ and $\mathcal{W}$ be a DL, a concept in L, a TBox for L and a witness for C, respectively. Then $\mathcal{W}$ is admissible w.r.t. $\mathcal{T}$ if, for each $x \in \Delta^{\mathcal{W}}$:*

$$\begin{array}{lll} C_1 \sqsubseteq C_2 \in \mathcal{T} & \textit{implies} & \neg C_1 \sqcup C_2 \in \mathcal{L}^{\mathcal{W}}(x), \\ C_1 \doteq C_2 \in \mathcal{T} & \textit{implies} & \neg C_1 \sqcup C_2 \in \mathcal{L}^{\mathcal{W}}(x) \textit{ and} \\ C_1 \doteq C_2 \in \mathcal{T} & \textit{implies} & C_1 \sqcup \neg C_2 \in \mathcal{L}^{\mathcal{W}}(x). \end{array}$$

A generalization of an *absorption* developed in [Horrocks and Tobies, 2000a,b] has been given in [Hudek and Weddell, 2006], dubbed *binary absorption*. We further extend binary absorption to accommodate nominals.

4.2.2 On Binary Absorption

Definition 29. ***Binary Absorption***. Let $\mathcal{K}$=$\{\mathcal{T}, \mathcal{A}\}$ be a KB. A *binary absorption* of $\mathcal{T}$ is a pair of TBoxes $(\mathcal{T}_u, \mathcal{T}_g)$ such that $\mathcal{T} \equiv \mathcal{T}_u \cup \mathcal{T}_g$ and $\mathcal{T}_u$ contains axioms of the form $A_1 \sqsubseteq C$, $\neg A_1 \sqsubseteq C$, $\exists S.\top \sqsubseteq C$ (resp. $\exists S^{-}.\top \sqsubseteq C$), and the form $(A_1 \sqcap A_2) \sqsubseteq C$ and $(\{a\} \sqcap A) \sqsubseteq C$, where $\{A, A_1, A_2\} \subseteq \mathrm{NC}$ and $a \in \mathrm{NI}$.

A binary absorption $(\mathcal{T}_u, \mathcal{T}_g)$ of $\mathcal{T}$ is called *correct* if it satisfies the following condition: For each witness $\mathcal{W}$ and $x \in \Delta^{\mathcal{W}}$, if all conditions in Figure 4.3 are satisfied, then $\mathcal{W}$ is admissible w.r.t. $\mathcal{T}$. A witness that satisfies the above property will be called *unfolded*. □

The distinguishing feature of this extension of binary absorption is the addition of the first four implications in Figure 4.3. Binary absorption itself allows additional axioms in $\mathcal{T}_u$ to

$$\begin{array}{rll}
\{a\} \in \mathcal{L}^{\mathcal{W}}(x) \text{ and } \{a\} \in \mathcal{L}^{\mathcal{W}}(y) & \text{implies} & x = y \\
\{\{a\}, A\} \subseteq \mathcal{L}^{\mathcal{W}}(x), \text{ and } (\{a\} \sqcap A) \sqsubseteq C \in \mathcal{T}_u & \text{implies} & C \in \mathcal{L}^{\mathcal{W}}(x) \\
(x, y) \in (S)^{\mathcal{I}} \text{ and } \exists S.\top \sqsubseteq C \in \mathcal{T}_u & \text{implies} & C \in \mathcal{L}^{\mathcal{W}}(x) \\
(x, y) \in (S)^{\mathcal{I}} \text{ and } \exists S^{-}.\top \sqsubseteq C \in \mathcal{T}_u & \text{implies} & C \in \mathcal{L}^{\mathcal{W}}(y) \\
\{A_1, A_2\} \subseteq \mathcal{L}^{\mathcal{W}}(x) \text{ and } (A_1 \sqcap A_2) \sqsubseteq C \in \mathcal{T}_u & \text{implies} & C \in \mathcal{L}^{\mathcal{W}}(x) \\
A \in \mathcal{L}^{\mathcal{W}}(x) \text{ and } A \sqsubseteq C \in \mathcal{T}_u & \text{implies} & C \in \mathcal{L}^{\mathcal{W}}(x) \\
\neg A \in \mathcal{L}^{\mathcal{W}}(x) \text{ and } \neg A \sqsubseteq C \in \mathcal{T}_u & \text{implies} & C \in \mathcal{L}^{\mathcal{W}}(x) \\
C_1 \sqsubseteq C_2 \in \mathcal{T}_g & \text{implies} & \neg C_1 \sqcup C_2 \in \mathcal{L}^{\mathcal{W}}(x) \\
C_1 \doteq C_2 \in \mathcal{T}_g & \text{implies} & \neg C_1 \sqcup C_2 \in \mathcal{L}^{\mathcal{W}}(x) \\
C_1 \doteq C_2 \in \mathcal{T}_g & \text{implies} & C_1 \sqcup \neg C_2 \in \mathcal{L}^{\mathcal{W}}(x)
\end{array}$$

Figure 4.3: Absorption witness conditions.

be dealt with in a deterministic manner. ABox absorption, treating assertions as axioms, extends binary absorption to handle nominals and to absorb domain and range constraints in a manner that resembles role absorption introduced in [Tsarkov and Horrocks, 2004].

Lemmas 4.2.3, 4.2.4 and 4.2.5 originally presented in [Horrocks and Tobies, 2000b] hold without modification. Lemma 4.2.6 shows that the generalized binary absorption is also a correct absorption.

Lemma 4.2.3. *Let $(\mathcal{T}_u, \mathcal{T}_g)$ be a correct binary absorption of $\mathcal{T}$. For any $C \in L$, C has a witness that is admissible w.r.t. $\mathcal{T}$ iff C has an unfolded witness.*

Lemma 4.2.4. *Let $\mathcal{T}$ be a primitive TBox and $\mathcal{T}_u$ defined as*

$$\{A \sqsubseteq C, \neg A \sqsubseteq \neg C \mid A \doteq C \in \mathcal{T}\}.$$

Then $(\mathcal{T}_u, \emptyset)$ is a correct absorption of $\mathcal{T}$.

Lemma 4.2.5. *Let $(\mathcal{T}_u, \mathcal{T}_g)$ be a correct absorption of a TBox $\mathcal{T}$.*

1. *If $\mathcal{T}'$ is an arbitrary TBox, then $(\mathcal{T}_u, \mathcal{T}_g \cup \mathcal{T}')$ is a correct absorption of $\mathcal{T} \cup \mathcal{T}'$.*

2. *If $\mathcal{T}'$ is a TBox that consists entirely of axioms of the form $A \sqsubseteq C$, where $A \in \mathrm{NC}$ and A is not defined in $\mathcal{T}_u$, then $(\mathcal{T}_u \cup \mathcal{T}', \mathcal{T}_g)$ is a correct absorption of $\mathcal{T} \cup \mathcal{T}'$.*

Lemma 4.2.6. *Let $(\mathcal{T}_u, \mathcal{T}_g)$ be a correct absorption of a TBox $\mathcal{T}$. If $\mathcal{T}'$ is a TBox that consists entirely of axioms of the form $(A_1 \sqcap A_2) \sqsubseteq C$ and $(\{a\} \sqcap A_3) \sqsubseteq D$, where $\{A_1, A_2, A_3\} \subseteq \mathrm{NC}$ and where none of A_1, A_2, A_3 are defined in $\mathcal{T}_u$, $a \in \mathrm{NI}$, then $(\mathcal{T}_u \cup \mathcal{T}', \mathcal{T}_g)$ is a correct absorption of $\mathcal{T} \cup \mathcal{T}'$.*

Proof. We only show the proof for axioms of the form $(A_1 \sqcap A_2) \sqsubseteq C$, and the proof of axioms of the form $(\{a\} \sqcap A_3) \sqsubseteq D$ follows, viewing $\{a\}$ as a primitive concept.

Observe that $\mathcal{T}_u \cup \mathcal{T}_g \cup \mathcal{T}' \equiv \mathcal{T} \cup \mathcal{T}'$ holds trivially. Let $C \in L$ be a concept and $\mathcal{W}$ be an unfolded witness for C w.r.t. the absorption $(\mathcal{T}_u \cup \mathcal{T}', \mathcal{T}_g)$. From $\mathcal{W}$, define a new witness $\mathcal{W}'$ for C by setting $\Delta^{\mathcal{W}'} = \Delta^{\mathcal{W}}$, $\cdot^{\mathcal{W}'} = \cdot^{\mathcal{W}}$, and defining $\mathcal{L}^{\mathcal{W}'}$ to be the function that, for every $x \in \Delta^{\mathcal{W}'}$, maps x to the set

$$\begin{aligned}\mathcal{L}^{\mathcal{W}}(x) \;&\cup\; \{\neg A_1, \neg A_2 \mid (A_1 \sqcap A_2) \sqsubseteq C' \in \mathcal{T}', \{A_1, A_2\} \cap \mathcal{L}^{\mathcal{W}}(x) = \emptyset\}\\ &\cup\; \{\neg A_1 \mid (A_1 \sqcap A_2) \sqsubseteq C' \in \mathcal{T}', A_1 \notin \mathcal{L}^{\mathcal{W}}(x), A_2 \in \mathcal{L}^{\mathcal{W}}(x)\}\\ &\cup\; \{\neg A_2 \mid (A_1 \sqcap A_2) \sqsubseteq C' \in \mathcal{T}', A_1 \in \mathcal{L}^{\mathcal{W}}(x), A_2 \notin \mathcal{L}^{\mathcal{W}}(x)\}.\end{aligned}$$

It is easy to see that $\mathcal{W}'$ is also unfolded w.r.t. the absorption $(\mathcal{T}_u \cup \mathcal{T}', \mathcal{T}_g)$. This implies that $\mathcal{W}'$ is also unfolded w.r.t. the (smaller) absorption $(\mathcal{T}_u, \mathcal{T}_g)$. Since $(\mathcal{T}_u, \mathcal{T}_g)$ is a correct absorption of $\mathcal{T}$, there exists an interpretation $\mathcal{I}$ stemming from $\mathcal{W}'$ such that $\mathcal{I} \models \mathcal{T}$. We show that $\mathcal{I} \models \mathcal{T}'$ also holds. Assume $\mathcal{I} \not\models \mathcal{T}'$. Then there is an axiom $(A_1 \sqcap A_2) \sqsubseteq C_1 \in \mathcal{T}'$ and an $x \in \Delta^{\mathcal{I}}$ such that $x \in ((A_1 \sqcap A_2))^{\mathcal{I}}$ but $x \notin (C_1)^{\mathcal{I}}$. By construction of $\mathcal{W}'$, $x \in ((A_1 \sqcap A_2))^{\mathcal{I}}$ implies $\{A_1, A_2\} \subseteq \mathcal{L}^{\mathcal{W}'}(x)$ because otherwise $\{\neg A_1, \neg A_2\} \cap \mathcal{L}^{\mathcal{W}'}(x) \neq \emptyset$ would hold in contradiction to (W3). Then, since $\mathcal{W}'$ is unfolded, $C_1 \in \mathcal{L}^{\mathcal{W}'}(x)$, which, again, by (W3), implies $x \in (C_1)^{\mathcal{I}}$, a contradiction.

Hence, we have shown that there exists an interpretation $\mathcal{I}$ stemming from $\mathcal{W}'$ such that $\mathcal{I} \models \mathcal{T}_u \cup \mathcal{T}' \cup \mathcal{T}_g$. By construction of $\mathcal{W}'$, any interpretation stemming from $\mathcal{W}'$ also stems from $\mathcal{W}$, hence $\mathcal{W}$ is admissible w.r.t. $\mathcal{T} \cup \mathcal{T}'$. □

4.3 A Procedure for ABox Absorption

This section presents a procedure for ABox absorption that works on $\mathcal{T}^i_{\mathcal{K}}, 0 \leq i \leq 2$, obtained from an initial $\mathcal{SHIQ}(\mathbb{D})$ knowledge base as shown in Section 4.1, extending binary absorptions with the following notable features, which

- maximally absorbs a TBox computed from any $\mathcal{SHIQ}(\mathbb{D})$ $\mathcal{K}$ (see Section 4.1), and
- retains all guarding constraints by prioritizing binary absorptions, and

- allows a DL reasoner to reason with restricted uses of nominals without incurring extra computational overhead, and
- makes it possible to absorb domain and range constraints in such a way that guards for domain and range restrictions become unnecessary.

The procedure is given in Section 4.3.1, which also serves as a general framework for absorption. Its correctness proof follows in Section 4.3.2.

4.3.1 The Procedure

The algorithm is given a TBox $\mathcal{T}$, that consists of arbitrary axioms. It proceeds by constructing five TBoxes $\mathcal{T}_g$, $\mathcal{T}_{prim}$, $\mathcal{T}_{uinc}$,, $\mathcal{T}_{binc}$, and $\mathcal{T}_{rinc}$ such that: $\mathcal{T} \equiv \mathcal{T}_g \cup \mathcal{T}_{prim} \cup \mathcal{T}_{uinc} \cup \mathcal{T}_{binc} \cup \mathcal{T}_{rinc}$, $\mathcal{T}_{prim}$ is primitive, $\mathcal{T}_{uinc}$ consists of axioms of the form $A_1 \sqsubseteq C$, $\mathcal{T}_{binc}$ consists of axioms of the form $A_1 \sqcap A_2 \sqsubseteq C$ and $\{a\} \sqcap A \sqsubseteq C$ and none of the above primitive concept are defined in $\mathcal{T}_{prim}$, and $\mathcal{T}_{rinc}$ consists of axioms of the form $\exists S.\top \sqsubseteq C$ (or $\exists S^-.\top \sqsubseteq C$). Here, $\mathcal{T}_{uinc}$ contains unary inclusion dependencies, $\mathcal{T}_{binc}$ contains binary inclusion dependencies and $\mathcal{T}_{rinc}$ contains domain and range inclusion dependencies.

In the first phase, the procedure moves as many axioms as possible from $\mathcal{T}$ into $\mathcal{T}_{prim}$. The procedure initializes $\mathcal{T}_{prim} = \emptyset$ and processes each axiom $X \in \mathcal{T}$ as follows.

1. If X is of the form $A \doteq C$, A is not defined in $\mathcal{T}_{prim}$, and $\mathcal{T}_{prim} \cup \{X\}$ is primitive, then move X to $\mathcal{T}_{prim}$.
2. If X is of the form $A \doteq C$, then remove X from $\mathcal{T}$ and replace it with axioms $A \sqsubseteq C$ and $\neg A \sqsubseteq \neg C$.
3. Otherwise, leave X in $\mathcal{T}$.

In the second phase, the procedure processes axioms in $\mathcal{T}$, either by simplifying them or by placing absorbed components in $\mathcal{T}_{uinc}$, $\mathcal{T}_{binc}$ or $\mathcal{T}_{rinc}$. $\mathcal{T}_g$ contains components that cannot be absorbed. We let $\mathbf{G} = \{C_1, \ldots, C_n\}$ represent the axiom $\top \sqsubseteq (C_1 \sqcup \ldots \sqcup C_n)$. Axioms are automatically converted to (out of) set notation. Recall that $\forall S.C$ (resp. $\forall S^-.C$) is considered a shorthand for $\exists^{\leq 0} S.\neg C$ (resp. $\exists^{\leq 0} S^-.\neg C$).

1. If $\mathcal{T}$ is empty, then return the binary absorption

$$(\{A \sqsubseteq C, \neg A \sqsubseteq \neg C \mid A \doteq C \in \mathcal{T}_{prim}\} \cup \mathcal{T}_{uinc} \cup \mathcal{T}_{binc} \cup \mathcal{T}_{rinc}, \mathcal{T}_g).$$

   Otherwise, remove an axiom $\mathbf{G}$ from $\mathcal{T}$.

2. Simplify $\mathbf{G}$.

   (a) If there is some $\neg C \in \mathbf{G}$ such that C is not an atomic concept, then add $(\mathbf{G} \cup \text{NNF}(\neg C) \setminus \{\neg C\}$ to $\mathcal{T}$, where the function NNF$(\cdot)$ converts concepts to negation normal form. Return to Step 1.

   (b) If there is some $C \in \mathbf{G}$ such that C is of the form $(C_1 \sqcap C_2)$, then add both $(\mathbf{G} \cup \{C_1\}) \setminus \{C\}$ and $(\mathbf{G} \cup \{C_2\}) \setminus \{C\}$ to $\mathcal{T}$. Return to Step 1.

   (c) If there is some $C \in \mathbf{G}$ such that C is of the form $C_1 \sqcup C_2$, then apply associativity by adding $(\mathbf{G} \cup \{C_1, C_2\}) \setminus \{C_1 \sqcup C_2\}$ to $\mathcal{T}$. Return to Step 1.

3. Partially absorb $\mathbf{G}$.

   (a) If $\{\neg\{a\}, \neg A\} \subset \mathbf{G}$, and A is a guard, then do the following. If an axiom of the form $(\{a\} \sqcap A) \sqsubseteq A'$ is in $\mathcal{T}_{binc}$, add $\mathbf{G} \cup \{\neg A'\} \setminus \{\neg\{a\}, \neg A\}$ to $\mathcal{T}$. Otherwise, introduce a new concept $A' \in$ NC, add $(\mathbf{G} \cup \{\neg A'\}) \setminus \{\neg\{a\}, \neg A\}$ to $\mathcal{T}$, and $(\{a\} \sqcap A) \sqsubseteq A'$ to $\mathcal{T}_{binc}$. Return to Step 1.

   (b) If $\{\neg A_1, \neg A_2\} \subset \mathbf{G}$, $(A_1 \sqcap A_2) \sqsubseteq A' \in \mathcal{T}_{binc}$, then add $\mathbf{G} \cup \{\neg A'\} \setminus \{\neg A_1, \neg A_2\}$ to $\mathcal{T}$. Return to Step 1.

   (c) If $\{\neg A_1, \neg A_2\} \subset \mathbf{G}$, and neither A_1 nor A_2 are defined in $\mathcal{T}_{prim}$, then do the following. If an axiom of the form $(A_1 \sqcap A_2) \sqsubseteq A'$ is in $\mathcal{T}_{binc}$, add $\mathbf{G} \cup \{\neg A'\} \setminus \{\neg A_1, \neg A_2\}$ to $\mathcal{T}$. Otherwise, introduce a new concept $A' \in$ NC, add $(\mathbf{G} \cup \{\neg A'\}) \setminus \{\neg A_1, \neg A_2\}$ to $\mathcal{T}$, and $(A_1 \sqcap A_2) \sqsubseteq A'$ to $\mathcal{T}_{binc}$. Return to Step 1.

   (d) If $\{\forall S.C\} = \mathbf{G}$ (resp. $\{\forall S^-.C\} = \mathbf{G}$), then do the following. Add $\exists S^-.\top \sqsubseteq C$ (resp. $\exists S.\top \sqsubseteq C$) to $\mathcal{T}_{rinc}$. Return to Step 1.

   (e) If $\forall S.\neg A$ (resp.$\forall S^-.\neg A$) $\in \mathbf{G}$, then do the following. Introduce a new internal primitive concept A' and add both $A \sqsubseteq \forall S^-.A'$ (resp. $A \sqsubseteq \forall S.A'$) and $(\mathbf{G} \cup \{\neg A'\}) \setminus \{\forall S.\neg A\}$ (resp. $\setminus\{\forall S^-.\neg A\}$) to $\mathcal{T}$. Return to Step 1.

4. Unfold $\mathbf{G}$. If, for some $A \in \mathbf{G}$ (resp. $\neg A \in \mathbf{G}$), there is an axiom $A \doteq C$ in $\mathcal{T}_{prim}$, then substitute $A \in \mathbf{G}$ (resp. $\neg A \in \mathbf{G}$) with C (resp. $\neg C$), and add $\mathbf{G}$ to $\mathcal{T}$. Return to Step 1.

5. Absorb $\mathbf{G}$. If $\neg A \in \mathbf{G}$ and A is not defined in $\mathcal{T}_{prim}$, add $A \sqsubseteq C$ to $\mathcal{T}_{uinc}$ where C is the disjunction of $\mathbf{G} \setminus \{\neg A\}$. Return to Step 1.

6. If none of the above are possible ($\mathbf{G}$ cannot be absorbed), add $\mathbf{G}$ to $\mathcal{T}_g$. Return to Step 1. □

In the above procedure, Step 3a is prioritized to ensure the pairing of a nominal and a guard for the purpose of guarded reasoning, which in addition guarantees that nominals never occur on the right-hand side of an axiom. Step 3b is performed before Step 3c to reduce nondeterminism of binary absorption and to minimize the number of fresh concepts to be introduced. In practice other heuristics may be applied for such purposes, for instance, a total ordering can be imposed on all concept names such that binary absorption absorbs axioms in specific ways.

4.3.2 Correctness

Termination of the procedure can be established by a counting argument. We now prove the correctness of the algorithm using induction. Lemmas 4.3.1 and 4.3.2 prove, in combination, that Steps 3a, 3b and 3c of the algorithm are correct. Lemmas 4.3.3 and 4.3.4 prove the correctness of Step 3d and Lemmas 4.3.5 and 4.3.6 prove the correctness of Step 3e, respectively. Note that Lemmas 4.3.2, 4.3.4, 4.3.5, and 4.3.6 are obtained directly from [Hudek and Weddell, 2006] without modification.

Lemma 4.3.1. *Let $\mathcal{T}_1, \mathcal{T}_2$, and $\mathcal{T}$ denote TBoxes, $C \in L$ an arbitrary concept, and A a primitive concept not used in C or $\mathcal{T}$. If $\mathcal{T}_1$ is of the form*

$$\mathcal{T}_1 = \mathcal{T} \cup \{(C_1 \sqcap C_2 \sqcap C_3) \sqsubseteq C_4\},$$

then C is satisfiable with respect to $\mathcal{T}_1$ iff C is satisfiable with respect to

$$\mathcal{T}_2 = \mathcal{T} \cup \{(C_1 \sqcap C_2) \sqsubseteq A, (A \sqcap C_3) \sqsubseteq C_4\}.$$

Lemma 4.3.2 proves that instead of introducing a new primitive concept every time Steps 3a, 3b and 3c of the algorithm are executed, a previously introduced atomic concept can be reused. **H** is written to denote an arbitrary axiom.

Lemma 4.3.2. *Let $\mathcal{T}_1, \mathcal{T}_2$, and $\mathcal{T}$ denote TBoxes, $C \in L$ an arbitrary concept, A a primitive concept not used in C or $\mathcal{T}$, and A_1, A_2, primitive concepts introduced by Steps 3a, 3b and 3c of our algorithm modified such that a new primitive is always introduced. If $\mathcal{T}_1$ is of the form*

$$\mathcal{T}_1 = \mathcal{T} \cup \{(C_1 \sqcap C_2) \sqsubseteq A_1, (C_1 \sqcap C_2) \sqsubseteq A_2\},$$

then C is satisfiable with respect to $\mathcal{T}_1$ iff C is satisfiable with respect to

$$\begin{aligned}\mathcal{T}_2 = &\{\mathbf{H} \textit{ where } A \textit{ is substituted for } A_1 \textit{ and } A_2 \mid \mathbf{H} \in \mathcal{T}\}\\ &\cup \{(C_1 \sqcap C_2) \sqsubseteq A\}\end{aligned}$$

Lemma 4.3.3. *Let $\mathcal{T}_1, \mathcal{T}_2$, and $\mathcal{T}$ denote TBoxes, $C \in L$ an arbitrary concept, and S a role. If $\mathcal{T}_1$ is of the form*

$$\mathcal{T}_1 = \mathcal{T} \cup \{\top \sqsubseteq \forall S.C_1\},$$

then C is satisfiable with respect to $\mathcal{T}_1$ iff C is satisfiable with respect to TBox

$$\mathcal{T}_2 = \mathcal{T} \cup \{\exists S^{-}.\top \sqsubseteq C_1\}.$$

Proof. First we prove the only-if direction. Assume C is satisfiable with respect to $\mathcal{T}_1$. For an interpretation $\mathcal{I} \in \mathrm{Int}(L)$ such that $\mathcal{I} \models \mathcal{T}_1$ and $(C)^{\mathcal{I}} \neq \emptyset$, we extend $\mathcal{I}$ to an interpretation $\mathcal{I}'$ such that $\mathcal{I}' \models \mathcal{T}_2$ and $(C)^{\mathcal{I}} \neq \emptyset$. First set $\mathcal{I}' = \mathcal{I}$. For each $x \in (\Delta)^{\mathcal{I}}$, we add x to $(\forall S.C_1)^{\mathcal{I}'}$. Then, $\mathcal{I}' \models \mathcal{T}_2$ and $(C)^{\mathcal{I}} \neq \emptyset$.

Now we prove the if direction. Assume C is satisfiable with respect to $\mathcal{T}_2$. For each interpretation $\mathcal{I} \in \mathrm{Int}(L)$ such that $\mathcal{I} \models \mathcal{T}_2$ and $(C)^{\mathcal{I}} \neq \emptyset$, it is also the case that $\mathcal{I} \models \mathcal{T}_1$. The proof is by contradiction. Assume $\mathcal{I} \not\models \mathcal{T}_1$. It must be the case that axiom $\top \sqsubseteq \forall S.C_1$ does not hold. Then for some $x \in (\Delta)^{\mathcal{I}}$, there is $y \in (\Delta)^{\mathcal{I}}$ such that $(x, y) \in S^{\mathcal{I}}$ and $y \notin (C_1)^{\mathcal{I}}$. However, from axiom $\exists S^{-}.\top \sqsubseteq C_1$ and the fact that $(y, x) \in (S^{-})^{\mathcal{I}}$, it follows that $y \in ((\exists S^{-}.\top))^{\mathcal{I}}$, thus, $y \in (C_1)^{\mathcal{I}}$: a contradiction. ☐

Lemma 4.3.4. *Let $\mathcal{T}_1, \mathcal{T}_2$, and $\mathcal{T}$ denote TBoxes, $C \in L$ an arbitrary concept, S a role and S^{-} an inverse role of S. If $\mathcal{T}_1$ is of the form*

$$\mathcal{T}_1 = \mathcal{T} \cup \{\top \sqsubseteq \forall S^{-}.C_1\},$$

then C is satisfiable with respect to $\mathcal{T}_1$ iff C is satisfiable with respect to TBox

$$\mathcal{T}_2 = \mathcal{T} \cup \{\exists S.\top \sqsubseteq C_1\}.$$

The proof of this lemma is similar to that of Lemma 4.3.3.

Lemma 4.3.5. *Let $\mathcal{T}_1, \mathcal{T}_2$, and $\mathcal{T}$ denote TBoxes, $C \in L$ an arbitrary concept, A a primitive concept not used in C or $\mathcal{T}$, and S a role. If $\mathcal{T}_1$ is of the form*

$$\mathcal{T}_1 = \mathcal{T} \cup \{\exists S.C_1 \sqsubseteq C_2\},$$

then C is satisfiable with respect to $\mathcal{T}_1$ iff C is satisfiable with respect to TBox

$$\mathcal{T}_2 = \mathcal{T} \cup \{C_1 \sqsubseteq \forall S^{-}.A, A \sqsubseteq C_2\}.$$

Lemma 4.3.6. *Let $\mathcal{T}_1, \mathcal{T}_2$, and $\mathcal{T}$ denote TBoxes, $C \in L$ an arbitrary concept, A a primitive concept not used in C or $\mathcal{T}$, and S a role. If $\mathcal{T}_1$ is of the form*

$$\mathcal{T}_1 = \mathcal{T} \cup \{\exists S^{-}.C_1 \sqsubseteq C_2\},$$

then C is satisfiable with respect to $\mathcal{T}_1$ iff C is satisfiable with respect to TBox

$$\mathcal{T}_2 = \mathcal{T} \cup \{C_1 \sqsubseteq \forall S.A, A \sqsubseteq C_2\}.$$

Theorem 4.3.7. *For any TBox $\mathcal{T}$, the ABox absorption algorithm computes a correct absorption of $\mathcal{T}$.*

Proof. The proof is by induction on iterations of our algorithm. We abbreviate the pair $(\{A \sqsubseteq C, \neg A \sqsubseteq \neg C \mid A \doteq C \in \mathcal{T}_{prim}\} \cup \mathcal{T}_{uinc} \cup \mathcal{T}_{binc} \cup \mathcal{T}_{rinc}, \mathcal{T}_g \cup \mathcal{T})$ as $\mathcal{T}$ and claim that this pair is always a correct binary absorption. Initially, $\mathcal{T}_{uinc}$, $\mathcal{T}_{binc}$, $\mathcal{T}_{rinc}$ and $\mathcal{T}_g$ are empty, primitive axioms are in $\mathcal{T}_{prim}$, and the remaining axioms are in $\mathcal{T}$. By Lemma 4.2.3, Lemma 4.2.4, Lemma 4.2.5, and Lemma 4.2.6, $\mathcal{T}$ is a correct absorption at the start of our algorithm. Assume we just finish iteration i and now perform iteration $i + 1$. By our induction hypothesis, $\mathcal{T}$ is a correct binary absorption after iteration i. We have a number of possible cases below.

- If we perform Step 3a, Step 3b or Step 3c then iteration $i+1$ is finished. Therefore, a newly introduced primitive concept only appears on the right hand side of an axiom once and Lemma 4.3.1 and Lemma 4.3.2 apply. We conclude that $\mathcal{T}$ is a correct binary absorption.

- If we perform Step 3d, then iteration $i+1$ is finished and by Lemma 4.3.3 and Lemma 4.3.4, $\mathcal{T}$ is a correct binary absorption.

- If we perform Step 3e, then iteration $i + 1$ is finished and by Lemma 4.3.5 and Lemma 4.3.6, $\mathcal{T}$ is a correct binary absorption.

- If we perform any of Steps 1, 2, 4, 5, or 6, then $\mathcal{T}$ is a correct binary absorption at the end of iteration $i+1$. This is because these steps use only equivalence preserving operations.

After the final iteration of our algorithm, $\mathcal{T}$ is a correct binary absorption by induction.

□

Once $\mathcal{T}^2_{\mathcal{K}}$ is generated, it can be supplied to the above absorption procedure to produce the final TBox $\mathcal{T}^3_{\mathcal{K}}$. By the correctness of the absorption procedure (Theorem 4.3.7), it follows that Theorem 4.1.2 is also applicable to $\mathcal{T}^3_{\mathcal{K}}$.

Example 8. ***Computing*** $\mathcal{T}^3_{\mathcal{K}}$ Applying the ABox absorption procedure given in this section to $\mathcal{T}^2_{\mathrm{univ}_{\mathcal{K}}}$ (see Example 7) then obtains the final absorbed terminology $\mathcal{T}^3_{\mathrm{univ}_{\mathcal{K}}}$ consisting of the following axioms:

Unary Absorptions	Binary Absorptions	General Axioms
$Dept \sqsubseteq \forall headOf^{-}.Prof$	$\{p\} \sqcap G \sqsubseteq Chair$	$\top \sqsubseteq Prof \sqcup G_{headOf}$
$Chair \sqsubseteq Prof$	$\{dept\} \sqcap G \sqsubseteq \exists headOf^{-}.\top$	
$Dept \sqsubseteq G_{headOf^{-}}$	$\{p\} \sqcap G \sqsubseteq \exists headOf.\top$	
	$\{p\} \sqcap G_{headOf} \sqsubseteq \exists headOf.(\{dept\} \sqcap G)$	
	$\{dept\} \sqcap G_{headOf^{-}} \sqsubseteq \exists headOf^{-}.\{p\}$	

Note that the general axiom $\top \sqsubseteq Prof \sqcup G_{headOf}$ could be absorbed by a unary absorption, for example, $\neg Prof \sqsubseteq G_{headOf}$, if negations on the left-hand side of an axiom are allowed. $\square$

4.4 An Empirical Evaluation

The empirical studies use an assertion retrieval engine, called CARE Assertion Retrieval Engine (CARE). CARE has an underlying DL reasoner that implements a standard tableau algorithm for $\mathcal{SHI}(\mathbb{D})$ knowledge bases. The DL reasoner features a limited number of optimizations, including ABox absorption, optimized double blocking in [Horrocks and Sattler, 2002], and dependency-directed backtracking in [Baader et al., 2003]. Moreover, the reasoner is also capable of reasoning with restricted ("safe" use of) nominals, where nominals appear only in binary axioms introduced in ABox absorption steps. At present, string is the only supported data type in the reasoner and a transitive closure algorithm is used to find clashes among concrete concepts [Nuutila, 1995]. The source code of the CARE system and the knowledge bases used in the empirical studies are publicly available.[2]

[2]http://code.google.com/p/care-engine/

4.4.1 Datasets and Queries

The experimental results in this chapter are mostly about query execution time. All times, given in seconds, were averaged out over *five* independent runs. A time out of 2000 seconds is set for the experiments, which were run on a single core of a 2.6GHz AMD Opteron 6282 SE processor on a Ubuntu 12.04 Linux server, with the Java heap set to 4GB. Some state-of-the-art DL reasoners will be used in comparing query execution time with CARE. In this situation, the latest release (at the time of writing) of these reasoner are used. Specifically, queries are posed via OWL API 3 for FaCT++ 1.6.2[3], Pellet 2.3.1[4] and HermiT 1.3.7[5], and via JRacer API for Racer 2.0[6] using the nRQL query language.

The DPC Datasets

A suite of datasets (KBs) about digital cameras has been built and used in the experiments. The KBs consist of digital camera model specifications extracted from `DPreview.com` and pricing information from *Google Product Search*. The ABox (data) of each KB contains a set of camera models described by a substantial number (around 70) of concrete feature concepts, in addition to other concepts. Every camera model has n $(0 \leq n \leq 10)$ products for sale through various sellers. The TBox (schema), being the same for all KBs, consists of 34 axioms, with the expressivity of $\mathcal{ALC}(\mathbb{D})$. Table 4.1 describes these the KBs, including the number of camera models (CMs), individuals (Inds), concept assertions (CAs) and role assertions (RAs) in each KB. Observe that DPC1 contains all the available digital cameras on `DPreview.com`, assuming there is only one product available for sale through each seller for a particular model. DPC2 correspondingly assumes there are two products for sale through each seller.

Queries for DPC datasets are shown in Table 4.2, which were designed to vary in query forms and selectivity. The answer set size of each query over every DPC KB is also given in Table 4.3. Specifically, Q1 is a selective query that retrieves a specific subclass of SLR cameras, which may be answered without assuming any hierarchy information. Q2, contrary to Q1, is less selective and must be answered using class hierarchy. Q3 is a selective query involving concrete facts. Q4 negates the concrete fact in Q3 to make it a range query. Q5, though in the form of an atomic concept (*Available_Digital_Camera*),

[3]http://code.google.com/p/factplusplus/

[4]http://clarkparsia.com/pellet/

[5]http://www.hermit-reasoner.com/

[6]http://www.racer-systems.com/

	CMs	Inds	CAs	RAs
HP	577	8774	4926	12101
Oly	1198	18354	10161	25705
DPC1	1895	28017	15445	39453
DPC2	1895	40595	15445	64510

Table 4.1: Descriptions of the DPC datasets.

Q1	$Digital\_SLR\_mirrorless$
Q2	$Compact\_Camera$
Q3	$Digital\_SLR \sqcap (user\_review = 5.00)$
Q4	$Digital\_SLR \sqcap (\neg(user\_review = 5.00))$
Q5	$Available\_Digital\_Camera$
Q6	$\exists hasManu.((manu\_name = \text{"}Kodak\text{"}) \sqcup (\exists locatedIn.Europe\_Country)))$
Q7	$(\exists hasInstance^{-}.(Lens\_mount = \text{"}Nikon\_F\_mount\text{"})) \sqcap (\exists hasSale.\exists hasSeller.(seller\_name = \text{"}Walmart\text{"}))$

Table 4.2: Sample queries posed over the DPC datasets.

it in fact uses an existential restriction as follows:

$$Available\_Digital\_Camera \equiv \exists hasSale.(\neg(inventory\_status = \text{"}outOfStock\text{"})).$$

Q6 has a disjunction occurring in the scope of an existential restriction. Q7 consists of a conjunction, of which the first (second) conjunct is a one-level (two-level) existential restriction involving concrete facts.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
HP	0	530	2	37	3737	8	0
Oly	1	1067	4	110	7906	183	3
DPC1	11	1673	7	181	12095	183	3
DPC2	11	1673	7	181	24190	183	6

Table 4.3: The answer set size of each query over the DPC KBs.

	FIs	Inds	CAs	RAs
USDA5	1205	1214	1214	12
USDA10	2994	3019	3019	583
USDA15	5259	5299	5299	649
USDA20	7295	7352	7353	1247
USDA25	8176	8250	8250	1535

Table 4.4: Descriptions of the USDA datasets.

The USDA Datasets

The USDA datasets, based on the the USDA Database [Service, 2012], provide composition data for 8176 food items. All food items are categorized into one of the 25 food groups, and each item may contain up to 146 food nutrient components (features). Each USDA KB includes a number of food groups, and each food item is modelled as an instance of its food group. Note that not all features are applicable for every food item. In the datasets, a small number of food items also have manufacturer information. The KBs are named such that the positive integer n implies the number of food groups included in the KBs.

The expressiveness of these KBs is $\mathcal{ALC}(\mathbb{D})$ and all KBs have the same set of axioms, which contains about 30 axioms. Most of the axioms are simple concept inclusions, while there are three equivalence axioms. Although the USDA KBs also have a multitude of concrete domain concepts for each instance, it differs from the DPC KBs in the size of role assertions. Table 4.4 lists information of the ABox part for each KB, including the number of food items, individuals, concept assertions, and role assertions, respectively.

The USDA KBs have smaller ABox than the DPC ones, particularly, role assertions are infrequent in the USDA KBs. For the USDA KBs, a set of eight queries were posed, as listed in Table 4.5. Specifically, Q1 retrieves instances of a specific food group which is included in every KB. Q2 uses an negation, while Q3 exploits concrete domain concepts. Q4 uses both a negation and a concrete domain concept. Q5 is more challenging as it involves a disjunctions and a negation. Q6 and Q7 both exploit existential restrictions, as well as a negated concrete domain concept. Note that Q8, though in the form of an atomic concept, is different from Q1 because it is defined by complex concepts as follows:

$$\begin{aligned} \mathit{ExcellentFoodDVProtein} &\equiv \mathit{ExcellentFoodDVProteinForMale} \\ &\quad \sqcup \mathit{ExcellentFoodDVProteinForFemale} \\ \mathit{ExcellentFoodDVProteinForMale} &\equiv \neg(\mathit{Protein\text{-}in\text{-}g} < 42) \\ \mathit{ExcellentFoodDVProteinForFemale} &\equiv \neg(\mathit{Protein\text{-}in\text{-}g} < 34.5) \end{aligned}$$

Thus, Q8 involves indefinite knowledge and concrete domain facts, while Q1 is a truly atomic concept. Finally, the answer set size of each query over every KB is also given in Table 4.6.

Q1	$Spices\_and\_Herbs$
Q2	$\neg Manufacturer$
Q3	$FOOD \sqcap (Total\_lipid\_fat = 0.00)$
Q4	$Spices\_and\_Herbs \sqcap \neg(Calcium\_Ca < 2000)$
Q5	$(Dairy\_and\_Egg\_Products \sqcup Soups\_Sauces\_and\_Gravies) \sqcap \neg(Vitamin\_C < 30)$
Q6	$\exists producedBy.(\neg(manu\_name = \text{“}Kraft\_Foods\_Inc.\text{”}))$
Q7	$\exists producedBy^{-}.(\neg(Energy < 100))$
Q8	$ExcellentFoodDVProtein$

Table 4.5: Sample queries posed over the USDA KBs.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
USDA5	61	1205	37	2	4	10	9	21
USDA10	61	2994	68	2	8	562	23	25
USDA15	61	5259	240	2	8	628	26	46
USDA20	61	7295	293	2	8	1224	44	85
USDA25	61	8176	299	2	8	1503	61	103

Table 4.6: The answer set size of each query over the USDA KBs.

LUBM Benchmark

The LUBM benchmark [Guo et al., 2005] consists of a TBox roughly in the DL dialect $\mathcal{SHI}(\mathbb{D})$, without using disjunctions. The ABox of LUBM can be generated on demand,

and the size of a LUBM ABox is identified by the number of universities. In these experiments, only one university of LUBM, denoted LUBM0, is used, which has about $20k$ concept assertions and $82k$ role assertions.

4.4.2 Comparing Guarding Strategies

Definition 25 effectively develops two types of guarding strategies, one being *partial guarding* (PG) and the other *full guarding* (FG). Intuitively, the PG strategy guards all individuals so that only individuals *relevant* to the individuals appearing in a query will be explored by reasoners. The FG strategy, in addition to PG, further guards concrete features so that only query-relevant feature concepts participate in reasoning. A naïve tableau-based implementation (the baseline approach), however, tends to explore all ABox individuals for each reasoning request, which is dubbed *no guarding* (NG).

The first experiments evaluated all queries over the HP KB under three different guarding strategies. The execution times of query answering under each strategy are given in Figure 4.4. CARE timed out after 2000 seconds under the NG strategy for all queries, because the baseline approach explores the whole ABox for each instance checking in the form of $\mathcal{K} \models a : C$ (note that CARE implements a "linear" instance retrieval for a given query concept). By adopting the PG method, CARE managed to answer all queries. Although most of the queries can be answered around 150 seconds, the query answering time for Q7 is close to the limit, requiring more than 1500 seconds. When the full guarding strategy is employed, the query response times for all queries have been substantially reduced to around 20 seconds, i.e., at least one order of magnitude faster for answering these queries than under the PG strategy. Thus, this set of experiments suggest the efficacy and the potential of the proposed guarding strategies. It should be noted that full guarding is extremely helpful for the knowledge base HP due to the sheer number of features available in the KB.

It has been demonstrated that ABox absorption can significantly improve instance checking over a no guarding strategy, i.e., the baseline approach that can be implemented by a tableau reasoner. However, most state-of-the-art reasoners develop a variety of optimization techniques at the level of answering *instance queries* (cf. Section 2.4.1). It might be argued that the effectiveness of guarding may be limited for highly optimized reasoners. To verify if ABox absorption offers additional benefits, the subsequent experiments were carried out to show CARE, which has a straightforward implementation of ABox absorption and *no other instance-level optimizations*, can indeed outperform highly optimized reasoners.

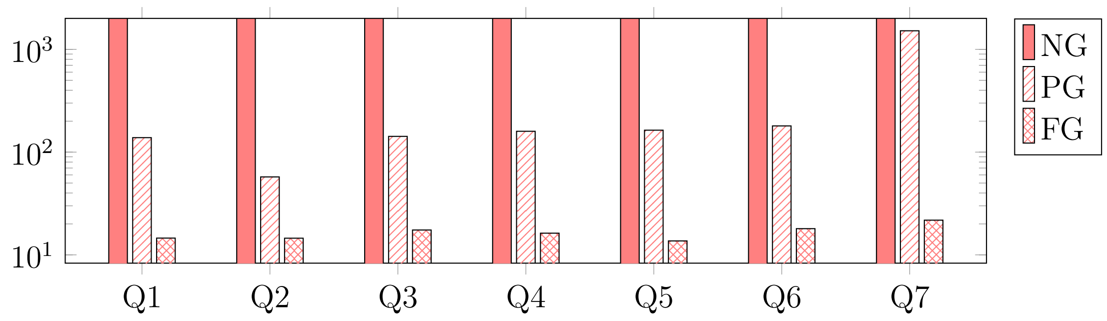


Figure 4.4: Comparing three guarding strategies in CARE over the HP dataset. NG, PG, and FG denote *no guarding*, *partial guarding*, and *full guarding*, respectively.

4.4.3 Comparing CARE with DL Reasoners

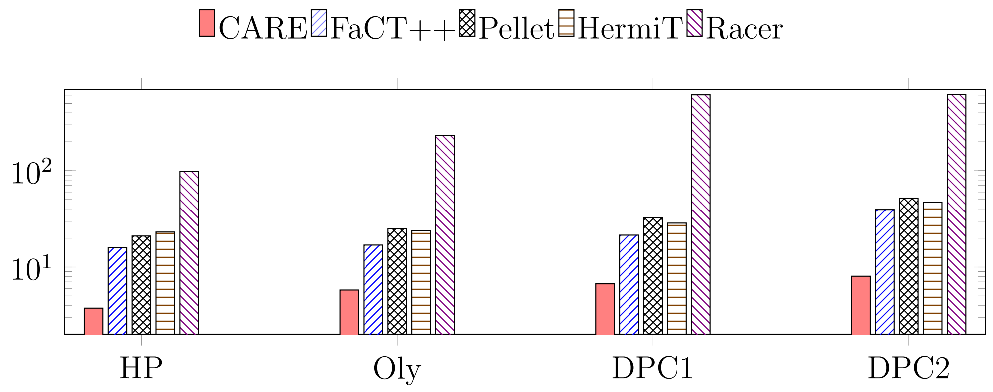
For highly optimized reasoners, numerous optimization techniques have been developed for answering instance queries. We juxtaposed the state-of-the-art reasoners with CARE in this section. The purpose is not to merely compare the performance of reasoners, but to validate the efficacy of ABox absorption for instance checking.

The experiments could have been performed for instance checking, however, OWL API does not provide a means to check instance checking problems; instead, instance queries were used in the experiments for comparison. It should be noted that most state-of-the-art DL reasoners have implemented many interesting optimizations for instance queries, while CARE has only implemented ABox absorption. We are also aware that some DL reasoners, such as Racer, Pellet, and HermiT, have more advanced optimizations for ground conjunctive query answering. Although some of the test queries may be converted into certain ground conjunctive queries that could be answered more efficiently by exploiting those particular optimizations, we did not consider such possibilities in these experiments because the purpose is to compare optimizations for instance queries (instance checking). Note that instance queries were posed as simple nRQL queries for Racer.

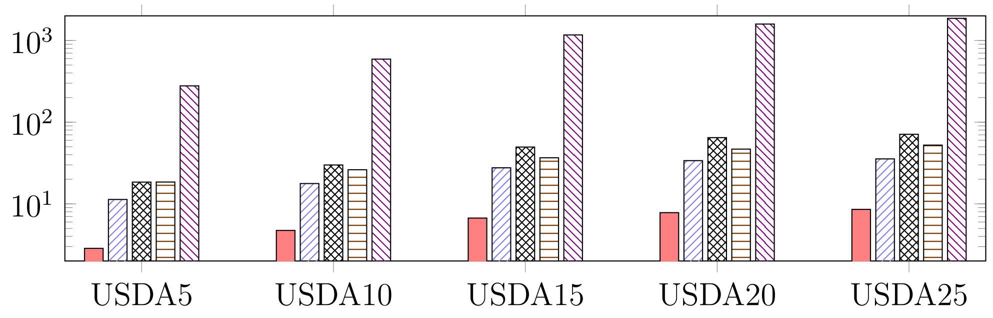
In the subsequent sections, we will investigate the preprocessing overhead of the DL reasoners over the aforementioned KBs, the query performance for different types of instance queries, and the effects of concrete domain concepts.

Preprocessing Overhead

Because existing optimized reasoners usually compute certain useful information prior to query answering, either in the KB loading phase or when explicitly requested, the times



(a) The DPC KBs.



(b) The USDA KBs.

Figure 4.5: Preprocessing times of the reasoners. Times are shown in logarithmic scale.

used for such preprocessing (and knowledge base loading) were compared separately from the actual query response times.

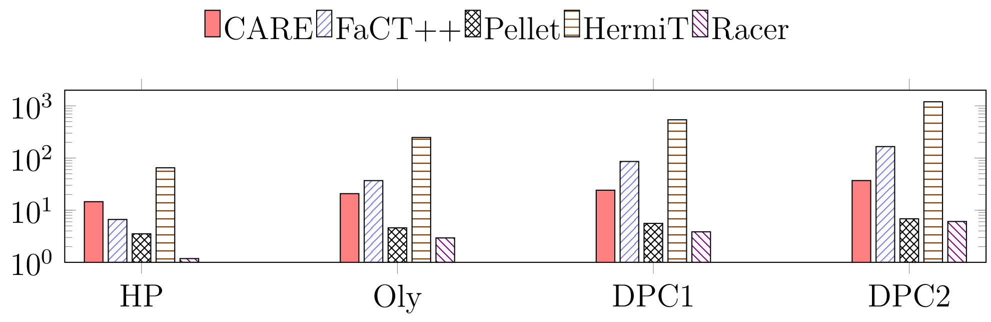
Figures 4.5a and 4.5b depict the average preprocessing times of each reasoner over the DPC and USDA KBs, respectively. It can be seen, in Figure 4.5a, that the system with the cheapest preprocessing was CARE, which spent no more 10 seconds in preprocessing, typically for loading the KBs, while Racer had the most expensive preprocessing phase, which was two orders of magnitude slower compared to CARE. The remaining reasoners had a more moderate preprocessing phase, which lasted 15-50 seconds. Similar to the results reported for the DPC KBs, CARE has the cheapest preprocessing overhead in Figure 4.5b, while FaCT++, Pellet, and HermiT are one order of magnitude slower. Racer required intensive preprocessing, which was two orders of magnitude slower than CARE. For the largest KB in this series, i.e., USDA25, Racer even needed more than 1800 seconds to preprocess it.

Query Performance

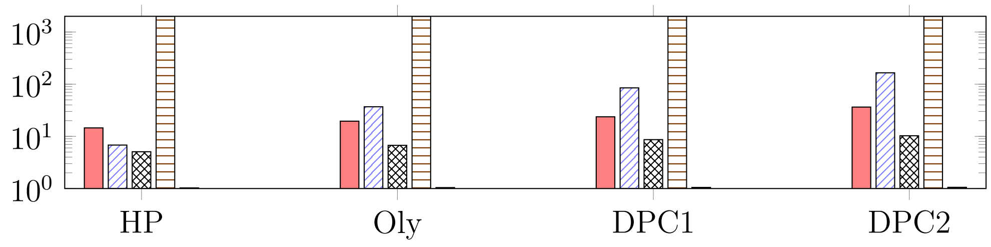
The next set of experiments was performed over the DPC and USDA knowledge bases. The details of these KBs and the test queries have been given in Section 4.4.1. Figure 4.6 and Figure 4.7 depict the query performance of all reasoners, with times presented in logarithmic scale, over the DPC and USDA KBs, respectively.

For the DPC KBs, as shown in Figure 4.6, Racer and Pellet had the best performance over all four KBs for Q1. Both of these two reasoners answered Q1 within 10 seconds, and Racer was more efficient. Given the amount of time for preprocessing, it is not surprising to observe Racer's performance. CARE, though four times slower than Racer and Pellet, was still faster than FaCT++ and HermiT over almost all KBs. Similar observations can be made for Q2 and Q3, of which both are positive instance queries. Note that HermiT already timed out after 2000 seconds for these positive queries. We conjecture that HermiT does not efficiently handle the large number of concrete domain concepts involved in reasoning. Q4 is a query that involves a negation of a concrete domain concept. In this case, CARE showed better query performance than all other reasoners when the size of KBs increase. Pellet, however, failed to retain quick query response times for this query. Indeed, Pellet was one order of magnitude slower than CARE, Racer and FaCT++. Recall that Q5 is defined by a complex concept which uses an existential restriction about a negative concrete domain concept. Pellet, HermiT, and Racer failed to answer this query for almost all the KBs within 2000 seconds. FaCT++ and CARE, on the other hand, continued to produce answers within a reasonable amount of time. In addition, CARE only took approximately one-third of the time as FaCT++ did. Q6 explicitly uses a disjunction. HermiT again did not manage to answer Q6 over any of the KBs. Racer succeeded in answering Q6 in the HP knowledge base only and timed out after 2000 seconds for the rest of the KBs. This is likely because Q6 contains a disjunction and the precomputed information in Racer does not aid in nondeterministic reasoning. CARE still performed better than FaCT++ as the KBs scale. Q7, the last query, only uses existential restrictions and inverse roles. Because no disjunctions are involved, Pellet restored its good performance. HermiT continued to time out, and the execution times of FaCT++ and Racer were comparable. CARE outperformed Racer and FaCT++, although it was about four times slower than Pellet.

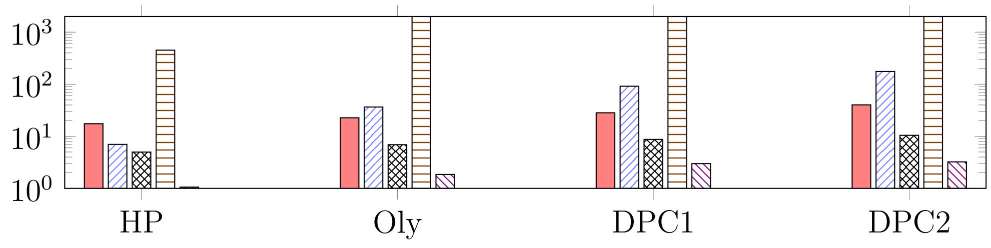
Similar observations can be made for query performance over the USDA KBs, as shown in Figure 4.7. For Q1 and Q2, only CARE and Racer were able to answer the queries within 15 seconds for all KBs. Racer was the most efficient in both cases since its precomputed indices facilitate answering queries about atomic concepts. FaCT++ and Pellet managed to answer the two queries with a few hundred seconds, however, the performance of both reasoners decreased dramatically as the size of KB increased. HermiT had poor performance



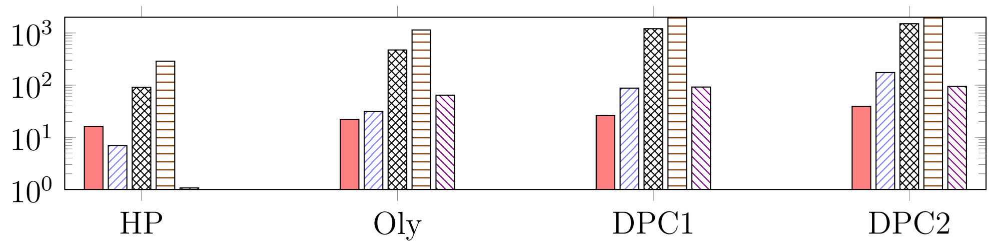
(a) Query 1.



(b) Query 2.

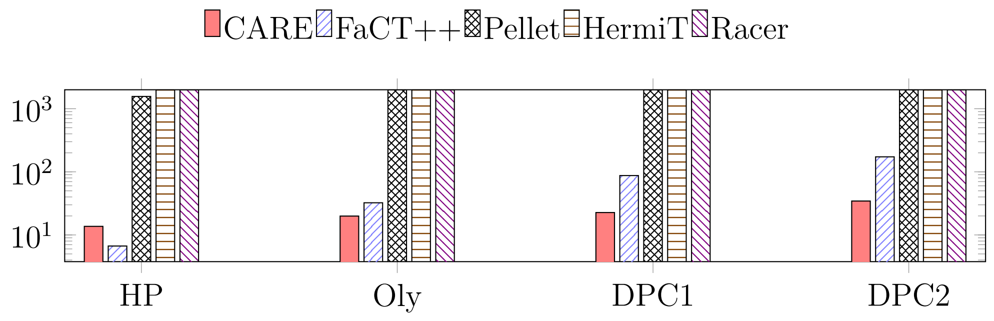


(c) Query 3.

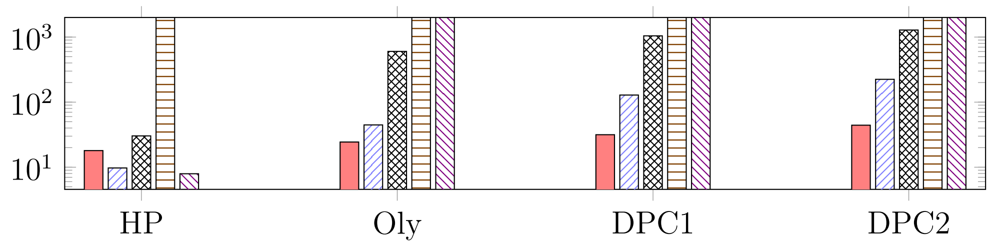


(d) Query 4.

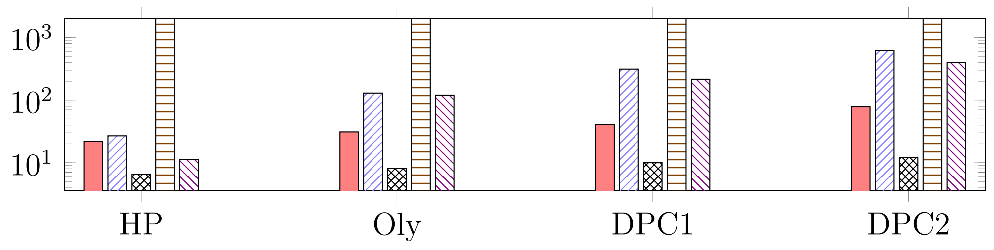
Figure 4.6: Comparing DL reasoners with CARE over the DPC KBs. Times are given in logarithmic scale.



(e) Query 5.



(f) Query 6.



(g) Query 7.

Figure 4.6: Comparing DL reasoners with CARE over the DPC KBs. Times are given in logarithmic scale.

in answering Q1 and timed out for most KBs when answering Q2. Q3 involves a concrete domain concept, which rendered Racer unable to answer the query within a reasonable period. It was likely due to a lack of precomputed information about concrete domain concepts in Racer. Instead, CARE was the most efficient reasoner in this case, which answered Q3 for all KBs with no more than 20 seconds. Interestingly, Pellet answered Q3 quite effectively in the smallest KB, but failed to retain the benefits while KBs increased in size. HermiT, however, did not answer Q3 within 2000 seconds even for USAD5. Similar observation can be made for Q4, because Q4 is structurally similar to Q3. It is easy to see that the results of Q3 and Q4 demonstrate the effects and advantages of full guarding in CARE. Similarly, CARE remained to be the most efficient for Q5. However, Pellet timed out for three KBS while answering Q5, because Q5 uses a disjunction. In the next query, Q6, the performance of Racer ad FaCT++ were comparable, while CARE was more efficient than them for larger KBs. Pellet, due to negations involved in Q6, did not perform well. For Q7, Racer became the most efficient, and answered Q7 within 10 seconds over all KBs. CARE still had very good performance and was one order of magnitude faster than FaCT++ and Pellet, except for USDA5. Q8 is a complexly defined concept that involves disjunctions and concrete domain concepts. CARE had the best performance, which answered Q8 within a few seconds, while the rest of the reasoners needed far more time. Note that HermiT was able to answer Q8 in some cases despite the disjunction and concrete domain concept.

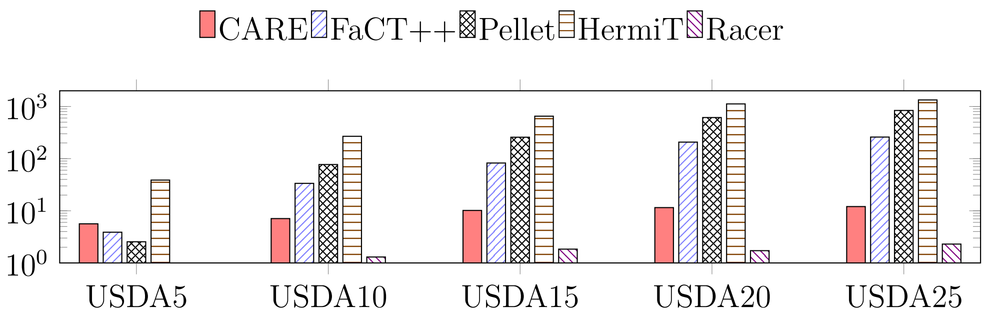
It can be seen that CARE and FaCT++ delivered the most consistent, good performance over these KBs, while CARE was generally two-folds faster than FaCT++ as the datasets scale. HermiT seemed to be unsuitable for instance queries over a large number of concrete domain concepts. Pellet and Racer were most efficient for positive queries, but occasionally showed unacceptable performance for non-positive queries, e.g., queries involving negations or disjunctions. Furthermore, Racer heavily relied on preprocessing to achieve good performance. Given that CARE is not as optimized as any of the other reasoners, the results are significant.

Query Performance and Concrete Features

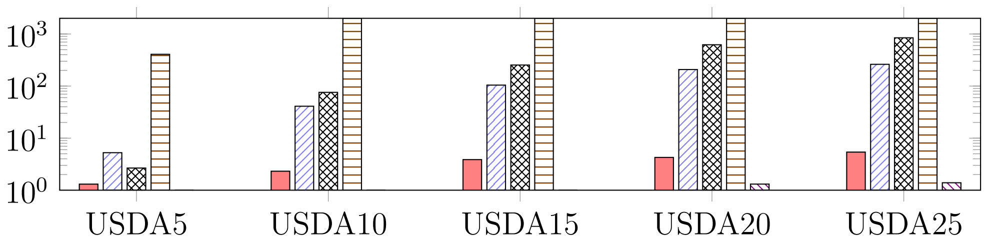
It can be argued that ABox absorption is efficient over the DPC and USDA KBs because a large number of concrete feature concepts are involved in query answering. In this section, a series of experiments on a series of synthetic knowledge bases without any concrete features were performed. The synthesized seed KB has two axioms and a number of ABox assertions replicated from the following pattern:

$$\begin{array}{ll} \text{TBox:} & (\exists R^{-}.\top \sqcap \exists S_1.\top) \sqsubseteq B,\ E \sqsubseteq (B \sqcup C) \\ \text{ABox:} & R(a_m, b_i),\ S_1(a_m, c_i),\ S_1(b_m, e_i),\ S_2(c_m, e_j), \\ & R^{-}(d_m, c_k),\ S_1(e_m, d_i),\ C(c_m)E(e_m). \end{array}$$

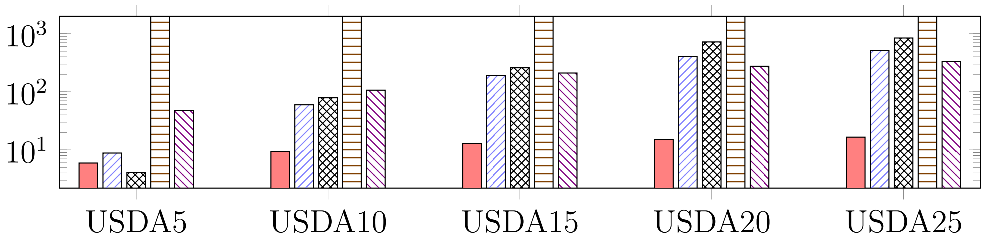
It is easy to see the DL dialect underlying these synthetic KBs is $\mathcal{ALCI}$. Initially, $m = 1000$ in the seed KB (syn1), which generates $5k$ instances, $6k$ role assertions and $2k$ concept assertions. Also observe that the second individuals in all role assertions were



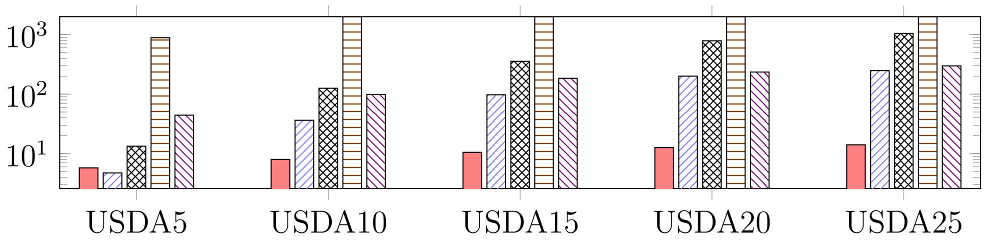
(a) Query 1.



(b) Query 2.

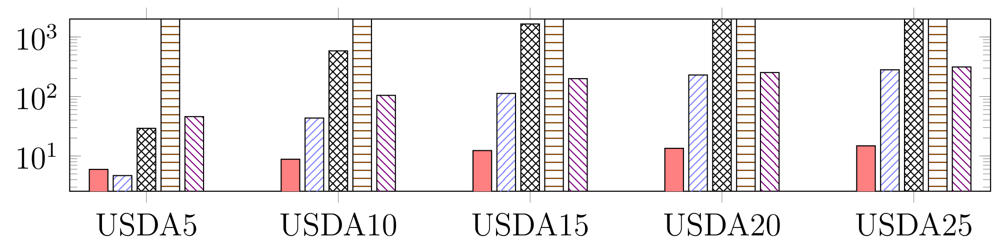


(c) Query 3.

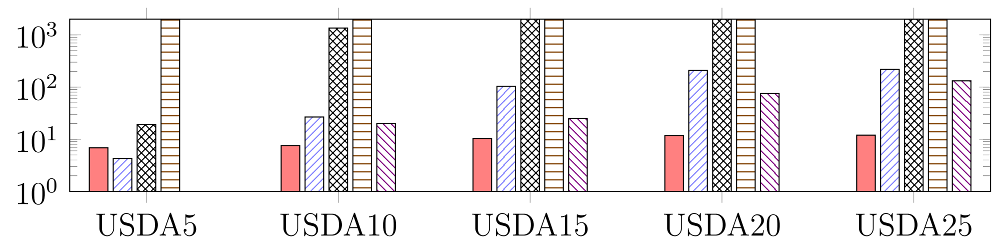


(d) Query 4.

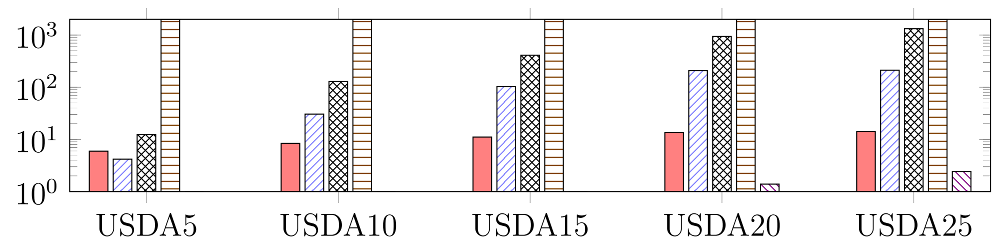
Figure 4.7: Comparing DL reasoners with CARE over the USDA KBs. Times are given in logarithmic scale.



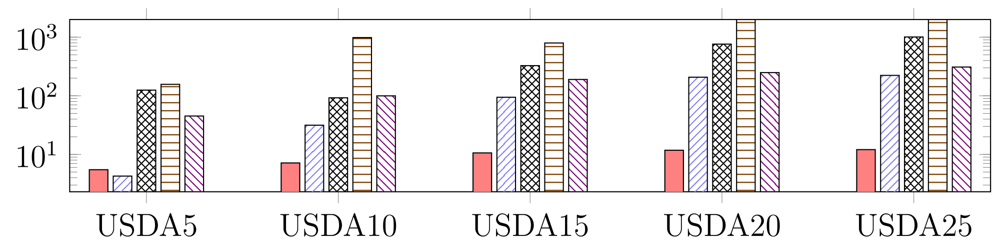
(e) Query 5.



(f) Query 6.

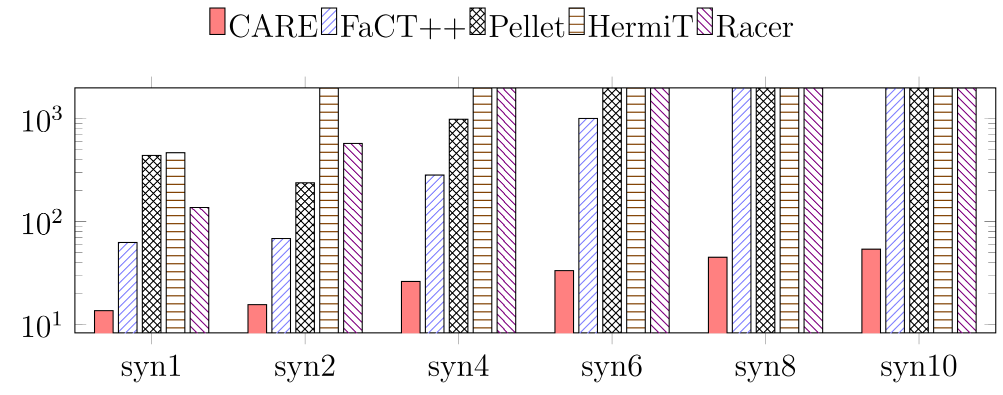


(g) Query 7.

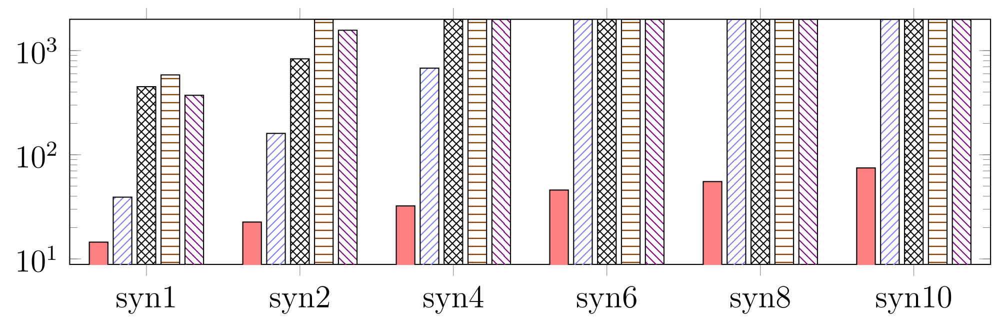


(h) Query 8.

Figure 4.7: Comparing DL reasoners with CARE over the USDA KBs. Times are given in logarithmic scale.



(a) Query 1: B.



(b) Query 2: $\exists R.B$.

Figure 4.8: Comparing query performance of CARE under partial guarding with other reasoners over the synthesized datasets.

randomly selected, e.g., $1 \leq i \leq m$. All other synthetical KBs were generated by repeating the seed KB k times, thus, named syn$k$.

Because these KBs contain no concrete facts, only partial guarding (PG) was adopted during reasoning. Figure 4.8 demonstrated that partial guarding (PG) alone can result in significant performance gains for state-of-the-art reasoners, in which the test queries are positive and very simple.

The results in Figure 4.8 are remarkable in that these highly optimized DL reasoners were, in most cases, at least one order of magnitude slower in answering simple instance queries (i.e., Q1 and Q2), compared to CARE. A study of the seed knowledge base can reveal that the ABox individuals form a number of relatively isolated conglomerates. The ABox absorption algorithm makes it possible for CARE to reason about individuals occurring in the same conglomerate while ignoring the rest of ABox. We conjecture that other reasoners had to include a large number of irrelevant individuals for reasoning, which resulted in inefficient query answering for simple queries such as B. In addition, the two queries take advantage of disjunctive knowledge in the TBox, which prevents reasoners from precomputing deterministic information to facilitate query answering. This set of experiments implies that ABox absorption is an essential optimization technique for instance checking over expressive DL knowledge bases.

The empirical studies in Section 4.4.3 demonstrated the efficacy of ABox absorption, i.e., guarded reasoning for instance checking. The comparison between CARE and other highly optimized DL reasoners suggests that ABox absorption is indispensable for *scalable instance query answering*, despite all the optimizations that have been designed for instance queries. Although reasoners often compute and cache some *deterministic* knowledge that can facilitate subsequent query answering, they lack efficient algorithms for querying *non-deterministic* knowledge, as shown by the performance of Pellet and Racer. ABox absorption has been shown extremely useful for reasoning with concrete domain concepts. We believe that, for example, HermiT can benefit from full guarding for circumstance where concrete domain concepts are prevalent in the ABox data. Even for reasoners with decent performance, such as FaCT++ and Pellet, ABox absorption can further optimize query answering for concrete domain concepts, e.g., Q3 and Q4, since it appeared that existing DL reasoners were more optimized for atomic concepts than for concrete domain concepts. Indeed, the nature of Web data makes it impossible to address efficient query answering under expressive constraints without concerning concrete domain concepts. Finally, since ABox absorption is complementary to all known query answering optimizations, existing reasoners can incorporate ABox absorption without incurring significant overhead in preprocessing (see Figures 4.5a and 4.5b). Similarly, the performance of CARE can be enhanced by adding optimizations implemented in other DL reasoners.

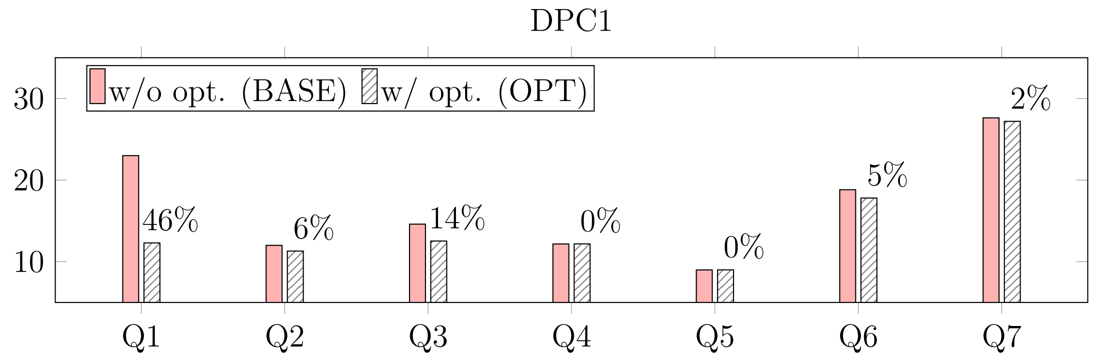
4.4.4 Optimizing Typing Constraints

Experiments in this section demonstrate the efficacy of the additional optimization applied to typing constraints of the form $L_1 \sqsubseteq \forall S.L_2$, in addition to the ordinary ABox absorption. The first set of experiments were performed on DPC1 using the queries listed in Figure 4.2. The second set of experiments were performed over the LUBM benchmark. Specifically, twelve queries out of the LUBM sample queries[7] were used (Q2 and Q9 were excluded since they are not expressible as instance queries) were tested over LUBM0. Since the experiments focus on instance checking, the selection conditions for

[7]`http://swat.cse.lehigh.edu/projects/lubm/queries-sparql.txt`

each of the twelve queries were reified, e.g., Q4 was rewritten as the instance query $\mathit{Professor} \sqcap \exists \mathit{worksFor}.A'$, for some fresh atomic concept A', and a new concept assertion (`http://www.Department0.University0.edu` : A') was added to the original ABox.

The results are listed in Figure 4.9 in which the execution time of CARE with optimized typing constraints (abbreviated as OPT) is compared with the execution time of CARE with the earlier version lacking this optimization (abbreviated as BASE). The percentage of improvement of the OPT method is also indicated above the bars.



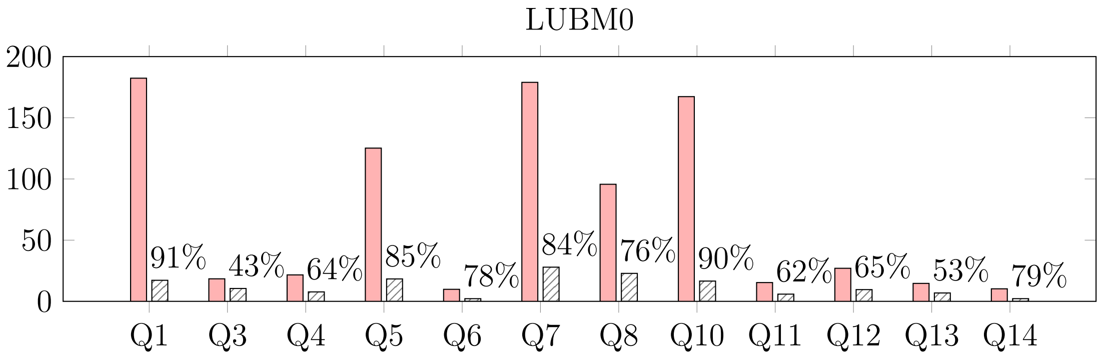


Figure 4.9: Performance gains in percentage when typing constraints are optimized.

For DPC1, the preprocessing times were about 7 seconds and 17 seconds for the BASE method and the OPT method, respectively. In the latter case, about 16% of the role assertions were optimized for one typing constraint (out of four); the remaining three typing constraints are for the same role. This limited use of typing constraints clearly makes the OPT method less beneficial to DPC1. Nevertheless, the evaluation of some instance queries has been improved. While four queries only witness a speedup of no more than 5% with OPT, there were queries that have been improved by over 40% (Q1). The limited gains are not surprising in view of the proportion of role assertions optimized.

For LUBM0, preprocessing cost 5 and 16 seconds with BASE and OPT, respectively. With the OPT method, about 23% of the role assertions were optimized for six typing constraints. Observe that, for each of the instance query, there is a dramatic improvement ranging from 43% to 91% with OPT. The results suggest that LUBM0 is more sensitive to typing constraints than DPC1, and that our method is most useful for ontologies that include a larger number of typing constraints, and where role assertion optimization nevertheless remains likely.

Another observation is that, for these knowledge bases, syntactic lookups in computing $\mathcal{T}^1_{\mathcal{K}}$ were not efficacious, that is, both $\mathcal{T}^{\forall}_{\rightarrow}$ and $\mathcal{T}^{\forall}_{\leftarrow}$ were empty (see Section 4.1.2). This explains why computing $\mathcal{T}^3_{\mathcal{K}}$ for both DPC1 and LUBM0 required more time than doing so with the BASE version. In particular, a large number of subsumption checks were performed during the computation of $\mathcal{T}^2_{\mathcal{K}}$ by the OPT method. Recall that the arc labelled "2" in Figure 4.1 indicates that some intermediate steps can be introduced to obtain $\mathcal{T}^2_{\mathcal{K}}$ in a way that can significantly reduce the number of such subsumption checks at load time.

This chapter presents a novel optimization, ABox absorption, for instance checking. This technique assumes the consistency of a knowledge base $\mathcal{K}$ and relies on binary absorption and a safe use of nominals to convert an ABox into a set of TBox axioms that contain guards (fresh concept names) to specify if the axioms can be fired by a tableau algorithm during expansion. This chapter also shows how to further advance this optimization by dropping guards partially for role assertions that satisfy all relevant local universal restrictions. The direct consequence of Abox absorption is the so-called guarded reasoning, which allows a tableau algorithm to explore only relevant individuals, instead of the whole ABox, for any given query concept. The experimental results of CARE on ABox absorption demonstrate the efficacy of this optimization and the usefulness of this method to support answering assertion queries.

Chapter 5

Query Compilation

This chapter shows how to compile a user query into an efficient query plan for query execution. In relational DBMSs, due to the large number of base relations to be considered, a query optimizer does not attempt to enumerate all possible query plans to find the optimal plan due to computational constraints; instead, the goal of the query compilation phase is to produce a *promising* query plan that is beneficial for query execution. For instance, consider the user query $(f = 1, g?)$ over a knowledge base $\mathcal{K}$, which retrieves $\mathcal{K}$ objects that satisfy $f = 1$ and return the description of g for the qualifying objects. Recall that Theorem 3.5.2 establishes a default plan for this user query, i.e., $\pi^{\mathcal{K}}_{g?}(\sigma^{\mathcal{K}}_{f=1}(P^{\mathcal{K}}))$. The default plan is expensive in the sense that reasoning w.r.t., $\mathcal{K}$ is used frequently for instance checking. However, this chapter shows that it is possible to obtain a range of more preferable query plans than the default plan by exploiting cached query results. For instance, consider a cached query result $S_1 = (f < 10, g?) :: f : \text{Ind}$, which stores explicitly the results of a user query that retrieves the descriptions of g for $\mathcal{K}$ objects satisfying $f < 10$ and sorts the results by the values of f and the object identifiers. Given S_1, it is possible to obtain the following plan for the previous user query: $S_1(f = 1)$, where S_1 is used for searching and $\mathcal{K}$ is not referenced in this plan.

In database terminology, a *logical* query plan, which refers to the algebraic expressions obtained by the application of a set of rewriting rules or heuristics, is distinguished from a *physical* query plan, which specifies actual algorithms for implementing operators in the algebraic expressions. In our context, these two notions coincide, i.e., *no distinctions are assumed between logical and physical plans.*

Query compilation consists of several phases to translate a user query into a query plan. DB2, for instance, implements the well-known *Query Graph Model* [Haas et al., 1989] that comprises sophisticated query analysis steps. Figure 5.1 depicts the overall query processing phases. The input to the compiler is a user query in the form of (C, Pd). After parsing and validation, the input query is fed into the plan generator, which applies a sequence of query rewriting rules to the user query. Because many query plans can be generated during this phase, there is also a component in this phase that searches among the plan space for the most promising query plan and outputs it. The final query plan is then executed by a plan interpreter and query answers are returned to users.

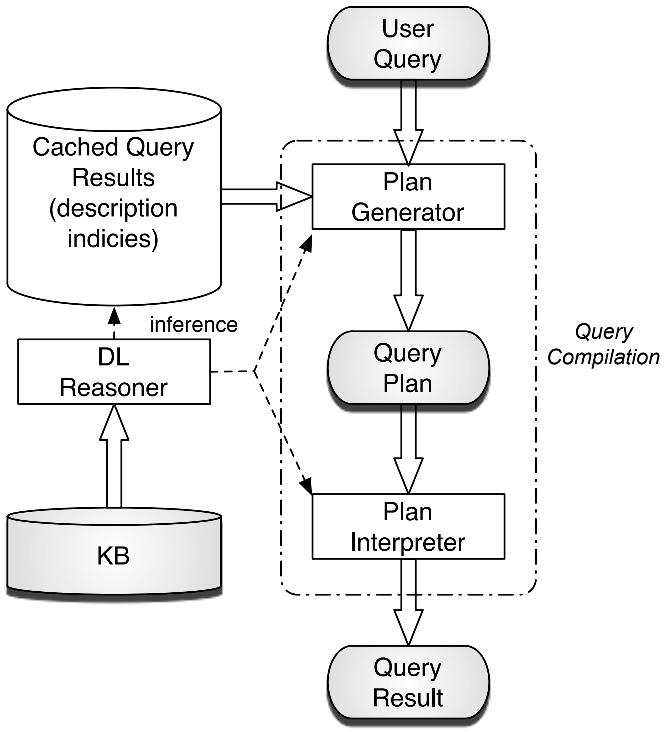


Figure 5.1: An overview of query processing for assertion retrieval.

This chapter is organized as follows. Section 5.1 defines a set of query rewriting rules that can be used to generate a variety of query plans for a user query (C, Pd). Section 5.2 addresses how to obtain a query plan in which the operators do not require any knowledge bases for reasoning. Finally, Section 5.3 elaborates the query compilation process, including the implementation of plan operators (Section 5.3.1) and the strategy of selecting the most promising plan in the plan space (Section 5.3.2).

5.1 Query Rewriting

Cached query results play an essential role in optimizing query answering performance, as seen in Section 5.1.1, where a suite of rewriting rules to translate a user query into equivalent algebraic forms is given that assume the availability of the required cached query results. This work does not consider which query results need to be materialized or how updates are handled, instead, any cached query results that participate in a query plan are assumed to be generated a priori.

Although not required, an input query of the form (C, Pd) can be parsed into a tree to ensure its syntactical correctness. The semantic correctness of a user query needs to be checked as well; for example, if the concept description C in (C, Pd) is semantically equivalent to $\bot$, the query is unsatisfiable and should retrieve no objects. The semantic correctness also applies to the projection descriptions. For instance, the query $(Book, \exists hasAuthor.publisher?)$ is syntactically correct, yet the feature *publisher* is likely not a valid attribute of authors in the underlying $\mathcal{K}$.

5.1.1 Queries as Algebraic Expressions

Recall that a valid user query can always be translated into an algebraic expression shown in (3.1). With the default query plan, various query plans can be obtained by applying a set of rewriting rules, as presented in [Pound et al., 2011]. Analogous to query optimization in relational databases, a set of relational-style rewriting rules (transformational laws) is first given in Figure 5.2. For clarity of presentation, the proofs of the lemmas and rewriting rules are given in Appendix A.

A cached query result corresponds to a *materialized view* in relational terms. The use of cached query results in query plans is analogous to *view-based query rewriting* in relational databases. The fundamental idea is to produce query plans that involve cached query results. A similar rewriting rule is given by Lemma 5.1.1. However, this lemma only concerns under what conditions a set of given cached query results can be used to replace the algebraic operator $P^{\mathcal{K}}$.

Lemma 5.1.1 (Cache Introduction). *Let (C, Pd) be a user query. The expression*

$$\pi^{\mathcal{K}}_{Pd}(\sigma^{\mathcal{K}}_{C}((S_1(C_1) \bowtie \cdots \bowtie S_n(C_n)))) \tag{5.9}$$

is equivalent to (3.1), provided that (i) $S_i = (D_i, Pd_i) :: Od_i$, (ii) $\mathcal{K} \models C \sqsubseteq (D_1 \sqcap \ldots \sqcap D_n)$, and (iii) $C_i = \coprod\{D \mid D \in \mathcal{L}_{Pd_i}, \mathcal{K} \models C \sqsubseteq D\}\rrbracket_{\mathcal{K}}$, for all $1 \leq i \leq n$.

$$\mathtt{Q}_1 \bowtie \mathtt{Q}_2 \leftrightarrow \mathtt{Q}_2 \bowtie \mathtt{Q}_1 \tag{5.1}$$
$$\mathtt{Q}_1 \bowtie (\mathtt{Q}_2 \bowtie \mathtt{Q}_3) \leftrightarrow (\mathtt{Q}_1 \bowtie \mathtt{Q}_2) \bowtie \mathtt{Q}_3 \tag{5.2}$$
$$S_1(\sigma_C^{\mathcal{K}}(\mathtt{Q})) \rightarrow \sigma_C^{\mathcal{K}}(S_1(\mathtt{Q})) \tag{5.3}$$
$$\sigma_{C_1}^{\mathcal{K}}(\sigma_{C_2}^{\mathcal{K}}(\mathtt{Q})) \leftrightarrow \sigma_{C_2}^{\mathcal{K}}(\sigma_{C_1}^{\mathcal{K}}(\mathtt{Q})) \tag{5.4}$$
$$\sigma_{C_1}^{\mathcal{K}}(\sigma_{C_2}^{\mathcal{K}}(\mathtt{Q})) \leftrightarrow \sigma_{C_1 \sqcap C_2}^{\mathcal{K}}(\mathtt{Q}) \tag{5.5}$$
$$\sigma_{C_1}^{\mathcal{K}}(\mathtt{Q}) \leftrightarrow \sigma_{C_2}^{\mathcal{K}}(\mathtt{Q}), \text{ if } \mathcal{K} \models C_1 \equiv C_2 \tag{5.6}$$
$$\pi_{Pd_1}^{\mathcal{K}}(\pi_{Pd_2}^{\mathcal{K}}(\mathtt{Q})) \leftrightarrow \pi_{Pd_1}^{\mathcal{K}}(\mathtt{Q}) \tag{5.7}$$
$$\pi_{Pd_1}^{\emptyset}(\pi_{Pd_2}^{\emptyset}(\mathtt{Q})) \leftrightarrow \pi_{Pd_1}^{\emptyset}(\mathtt{Q}), \text{ if } \mathcal{L}_{Pd_1} \subseteq \mathcal{L}_{Pd_2} \tag{5.8}$$

Figure 5.2: A list of relational-style rewriting rules.

Proof. See Appendix A. □

Because $P^{\mathcal{K}}$ can potentially computes a very large set of objects, i.e., all the objects in $\mathcal{K}$, which is the input for the subsequent, computationally expensive selection and projection operations. With the use of cached query results in Lemma 5.1.1, the size of the input to the selection operation is smaller, and it is possible to apply other rewriting rules (as discussed later in this section) to the cached query results to obtain query plans that reduce or avoid the use of reasoning w.r.t. $\mathcal{K}$. Note, again, the availability, the usability, and the maintenance of cached query results are not addressed in this work.

The introduction of cache scans in place of the primary operation, as in Lemma 5.1.1, may be necessary in some situations, for example, when $\mathcal{K}$ is not accessible or it contains a large number of individuals. Note that (5.9) permits a multi-way join instead of binary joins, because, to this end, the order of joins does not matter. However, the join order may affect subsequent rewriting. There are various ways to determine a suitable join order of cache scans, and one will be discussed in Section 5.3. The next rule, (5.10), demonstrates the possibility to remove a selection operator.

Lemma 5.1.2 (Removing Redundant Selections). *The selection operation $\sigma_C^{\mathcal{K}}(\cdot)$ can be removed from (5.9) to obtain the expression*

$$\pi_{Pd}^{\mathcal{K}}((S_1(C_1) \bowtie \cdots \bowtie S_n(C_n))) \tag{5.10}$$

if $\mathcal{K} \models (C_1 \sqcap \ldots \sqcap C_n) \sqsubseteq C$.

Proof. See Appendix A. □

Lemma 5.1.3 below allows one to nest cache scans such that a join of two cache scans can be pushed inside one cache scan. For this rewriting rule, the order to join two cache scans impacts the final plan (see Example 9 for an illustration). Note the condition on Pd, the projection description to be introduced for the newly nested cache scan. In particular, $Pd = \top?$ always satisfies the condition. This rule is particularly useful if $S_j(C_j)$ produces complicated concept assertions, for example, $(a : D_1 \sqcap \cdots \sqcap D_m)$ for some individual a and a large integer m. In this case, $\pi^{\mathcal{K}}_{Pd}(S_j(C_j))$ would produce concept assertions of the form $(a : \top)$, which makes the outmost projection much easier to compute.

Lemma 5.1.3 (Nested Searches)**.** *An expression of the form*

$$\pi^{\mathcal{K}}_{Pd'}(S_i(C_i) \bowtie S_j(C_j))$$

is equivalent to

$$\pi^{\mathcal{K}}_{Pd'}(S_i(C_i \bowtie \pi^{\mathcal{K}}_{Pd}(S_j(C_j))))$$

where Pd is such that $\mathcal{L}_{Pd} \subseteq \mathcal{L}_{Pd_i}$ (the projection description used to define S_i) and Pd' is an arbitrary projection.

Proof. See Appendix A. □

5.2 $\mathcal{K}$-free Rewriting

There are a variety of cases in which the operators in the algebra can be evaluated without general DLs reasoning. This section allows us to determine whether a particular algebraic operator does not refer to the knowledge base $\mathcal{K}$. This goal amounts to finding conditions under which: $\mathsf{OP}^{\mathcal{K}}(\mathtt{Q}) = \mathsf{OP}^{\emptyset}(\mathtt{Q})$ for an operator OP in the algebra and every knowledge base $\mathcal{K}$ (where $\mathsf{OP}^{\emptyset}$ denotes evaluating the operator with respect to the empty knowledge base). Intuitively this means that the concept assertions in the answers to $\mathtt{Q}_i$ contain sufficient amount of *explicit* information about an individual; the goal is to ultimately obtain this information using some cached query results rather than via general reasoning in $\mathcal{K}$. In assertion retrieval algebra, this degenerates to optimizing the operators $\sigma^{\mathcal{K}}_{C}(\mathtt{Q})$ and $\pi^{\mathcal{K}}_{Pd}(\mathtt{Q})$ such that $\mathcal{K}$ can be substituted by $\emptyset$. The leaf operator $P^{\mathcal{K}}$ always requires an underlying $\mathcal{K}$ to obtain all the instance names; therefore, it can be substituted by one or more cached query results to obtain a $\mathcal{K}$-free query plan. Such a translation is useful when the entire $\mathcal{K}$ is unavailable.

Definition 30. ***Representative Language***. A language of representative concepts for an algebraic query $\mathtt{Q}$. denoted $\mathcal{L}_{\mathtt{Q}}$, is defined as follows:

$$\mathcal{L}_{\mathtt{Q}} = \begin{cases} \{C\} & \text{if } \mathtt{Q} = \text{“}C\text{”;} \\ \{\top\} & \text{if } \mathtt{Q} = \text{“}P^{\mathcal{K}}\text{”;} \\ \mathcal{L}_{Pd_i} & \text{if } \mathtt{Q} = \text{“}S_i(\mathtt{Q}_1)\text{”;} \\ \mathcal{L}_{\mathtt{Q}_1} & \text{if } \mathtt{Q} = \text{“}\sigma^{\mathcal{K}}_{C}(\mathtt{Q}_1)\text{”;} \\ \mathcal{L}_{Pd} & \text{if } \mathtt{Q} = \text{“}\pi^{\mathcal{K}}_{Pd}(\mathtt{Q}_1)\text{”; and} \\ \{C \sqcap D \mid \; C \in \mathcal{L}_{\mathtt{Q}_1}, D \in \mathcal{L}_{\mathtt{Q}_2}\} & \text{if } \mathtt{Q} = \text{“}\mathtt{Q}_1 \bowtie \mathtt{Q}_2\text{”.} \end{cases}$$

□

The following corollary derives from Lemma 3.2.1, which establishes a useful property for the representative language $\mathcal{L}_{\mathtt{Q}}$ of any pure $\mathtt{Q}$. It is easy to see that Corollary 5.2.1 also holds if $\mathcal{L}_{\mathtt{Q}}$ is replaced by $\mathcal{L}^{\text{fin}}_{\mathtt{Q}}$.

Corollary 5.2.1. *For any pure $\mathtt{Q}$, any pair of concepts $\{D_1, D_2\} \subseteq \mathcal{L}_{\mathtt{Q}}$ and $\models D_1 \sqcap D_2 \not\sqsubseteq \bot$ (i.e., $D_1 \sqcap D_2$ is a satisfiable concept), there is $D_3 \in \mathcal{L}_{\mathtt{Q}}$ such that $\models D_3 \equiv D_1 \sqcap D_2$.*

Proof. Considering any pure query plan $\mathtt{Q}$, the claim holds vacuously for $\mathtt{Q} = P^{\mathcal{K}}$. The claim derives directly from Lemma 3.2.1 when $\mathtt{Q} = S_i(\mathtt{Q}_1)$ and $\mathtt{Q} = \pi^{\mathcal{K}}_{Pd}(\mathtt{Q}_1)$. When $\mathtt{Q} = \sigma^{\mathcal{K}}_{C}(\mathtt{Q}_1)$, the claim holds by induction hypothesis because $\mathtt{Q}_1$ is also pure. Now we elaborate the last case, i.e, $\mathtt{Q} = \mathtt{Q}_1 \bowtie \mathtt{Q}_2$. Assuming any pair $\{D_1, D_2\} \subseteq \mathcal{L}_{\mathtt{Q}}$, let $D_1 = D^1_1 \sqcap D^2_1$ and $D_2 = D^1_2 \sqcap D^2_2$, where $D^1_i \in \mathcal{L}_{\mathtt{Q}_1}$ and $D^2_i \in \mathcal{L}_{\mathtt{Q}_2}, i \in \{1, 2\}$. Note that $D_1 \sqcap D_2$ is satisfiable, so any subconcepts in D_1 and D_2 must be satisfiable. By induction hypotheses on the two pure query plans $\mathtt{Q}_1$ and $\mathtt{Q}_2$, it follows that there are $D^1_3 \in \mathcal{L}_{\mathtt{Q}_1}$ such that $\models D^1_3 \equiv D^1_1 \sqcap D^1_2$ and $D^2_3 \in \mathcal{L}_{\mathtt{Q}_2}$ such that $\models D^2_3 \equiv D^2_1 \sqcap D^2_2$. Therefore, $D^1_3 \sqcap D^2_3 \in \mathcal{L}_{\mathtt{Q}}$; moreover, $\models D^1_3 \sqcap D^2_3 \equiv D^1_1 \sqcap D^1_2 \sqcap D^2_1 \sqcap D^2_2 \equiv D_1 \sqcap D_2$. □

Given the definition of the representative language of a query plan $\mathtt{Q}$, we can establish the following correspondence for testing $\mathcal{K}$-free rewriting. Recall that the semantic subset relation $\hookrightarrow$ is given in Definition 15.

Theorem 5.2.2. *Let (C, Pd) be a user query, (C_i, Pd_i) queries that define cached query results S_i, and $\mathtt{Q}$ a query plan equivalent to (C, Pd). Then, for every* pure *subquery $\mathtt{Q}'$ of $\mathtt{Q}$, the following holds:*

1. $[\![\sigma_C^{\mathcal{K}}(\mathtt{Q}')]\!]_{\mathcal{K}}^{\mathbb{SI}} = [\![\sigma_C^{\emptyset}(\mathtt{Q}')]\!]_{\emptyset}^{\mathbb{SI}}$ *iff* $\{C\} \hookrightarrow \mathcal{L}_{\mathtt{Q}'}$;
2. $[\![\pi_{Pd}^{\mathcal{K}}(\mathtt{Q}')]\!]_{\mathcal{K}}^{\mathbb{SI}} = [\![\pi_{Pd}^{\emptyset}(\mathtt{Q}')]\!]_{\emptyset}^{\mathbb{SI}}$ *iff* $\mathcal{L}_{\pi_{Pd}^{\mathcal{K}}(\mathtt{Q}')} \hookrightarrow \mathcal{L}_{\mathtt{Q}'}$.

Proof. See Appendix A. □

The above theorem becomes an effective rewriting rule when finite approximations of the languages $\mathcal{L}_Q$ are defined. The precondition for the rewriting defined in Theorem 5.2.2 can then be tested using finitely many subsumption tests in the underlying description logic.

Definition 31. ***Finite Approximation***. Let (C, Pd) be a user query and $S_i = (C_i, Pd_i)$ cached query results. The set of admissible concrete domain values (strings), $S_{\mathbb{D}}$, is defined to be the set of values k such that $(f = k)$, for some feature f, is a subexpression in the user query or S_i. $\mathcal{L}_{\mathtt{Q}}^{\text{fin}}$ is a finite approximation of $\mathcal{L}_{\mathtt{Q}}$ if for any concept D that appears in $\mathcal{L}_{\mathtt{Q}}^{\text{fin}}$, whenever $f = k'$ is a subconcept of D, it holds that $k' \in \{*\} \cup S_{\mathbb{D}}$, where $*$ is an arbitrary concrete domain value (string) and $* \notin S_{\mathbb{D}}$. □

Lemma 5.2.3. *$\mathcal{L}_{Q}^{\text{fin}}$ is finite; $\mathcal{L}_{\mathtt{Q}_1} \hookrightarrow \mathcal{L}_{\mathtt{Q}_2}$ iff $\mathcal{L}_{\mathtt{Q}_1}^{\text{fin}} \hookrightarrow \mathcal{L}_{\mathtt{Q}_2}^{\text{fin}}$.*

Proof. See Appendix A. □

Example 9 illustrates how a user query can be transformed via applications of the query rewriting rules presented in this section.

Example 9. ***Query Rewriting*** Consider a knowledge base $\mathcal{K}$ about cameras and three cached query results computed w.r.t. it. The definitions of these cached query results are given below.

$$\begin{aligned}
S_1 &= (\top, \mathit{user\_review}?) :: \mathit{user\_review} : \text{Ind}\\
S_2 &= (\mathit{release\_date} < 20111001, \mathit{release\_date}?) :: \mathit{release\_date} : \text{Ind}\\
S_3 &= (\top, \mathit{resolutions}?) :: \text{Un}
\end{aligned}$$

Suppose now the following user query is issued:

$$((\mathit{user\_review} = 4.50) \sqcap (\mathit{release\_date} = 20110105), \mathit{resolutions}?) \tag{5.11}$$

Specifically, (5.11) retrieves all objects released on a specific date with a fixed user review rating. Let C_1, C_2 denote $(\mathit{user\_review} = 4.50)$ and $(\mathit{release\_date} = 20110105)$, respectively.

Figure 5.3 lists a number of query plans that can be obtained by applying the appropriate query rewriting rules.

Plan 5.3a is the default plan obtained by (3.1). Plan 5.3b applies (5.9) to the default plan using the three cached query results defined above. Note that in this query plan the join order is not assumed. By removing the selection operation, i.e., applying the rewriting rule (5.10), plan 5.3c is obtained. Next, nested searches can be enabled, as Lemma 5.1.3 states. Plan 5.3d indeed assumes the following join order for the three cache scans: $\pi^{\mathcal{K}}_{\top ?}(\cdot) \bowtie (S_1(C_1) \bowtie S_2(C_2))$. Of course, a different order can be assumed, but the subsequent plans will be different. In particular, the join order assumed by plan 5.3d has a benefit that other orders do not have, as discussed later. Plan 5.3e differs from plan 5.3d only in the use of $\mathcal{K}$. In fact, plan 5.3e is a $\mathcal{K}$-free query plan, the focal interest of Section 5.2. Plan 5.3f is obtained by two observations from the previous plan 5.3e. First, the projection description in $\pi^{\emptyset}_{resolutions?}(\cdot)$ is the same as the one used for defining S_3. In this case, the projection operation can be dismissed. Second, the join order chosen by plan 5.3d permits the simplification of the first join in plan 5.3e: $\top$ is included in the concept of every concept assertion produced by $\pi^{\emptyset}_{\top ?}(\cdot)$, so, the operation $\pi^{\emptyset}_{\top ?}(\cdot) \bowtie \top$ can be simplified as $\pi^{\emptyset}_{\top ?}(\cdot)$. □

5.2.1 Determining $\mathcal{K}$-free Rewritings

Lemma 5.2.3 restricts the constants to those occurring in the queries, making it possible to determine $\mathcal{K}$-free query rewriting. In the case of rewriting $\sigma^{\mathcal{K}}_{C}(\mathtt{Q}')$ into $\sigma^{\emptyset}_{C}(\mathtt{Q}')$, it suffices to show $\{C\} \hookrightarrow \mathcal{L}^{\text{fin}}_{\mathtt{Q}'}$. However, when $\mathcal{L}^{\text{fin}}_{\mathtt{Q}'} = \mathcal{L}_{Pd}$ for some Pd, deciding the relationship $\hookrightarrow$ seems infeasible due to the size of $\mathcal{L}_{Pd}$. It is possible to use C to eliminate obviously unqualified concepts in $\mathcal{L}_{Pd}$; for example, any concept $D \in \mathcal{L}^{\text{TUP}}_{Pd}$ that fails to satisfy $\models C \sqsubseteq D$ can be eliminated because $\models C \sqsubseteq D_1 \sqcap D_2$ requires $\models C \sqsubseteq D_1$ and $\models C \sqsubseteq D_2$. For queries in the form of $\pi^{\mathcal{K}}_{Pd}(\mathtt{Q}')$, deciding $\mathcal{K}$-free rewriting is more challenging. To show $\mathcal{L}_{Pd} \hookrightarrow \mathcal{L}^{\text{fin}}_{\mathtt{Q}'}$, computing the left-hand side $\mathcal{L}_{Pd}$ can be reduced to computing $\mathcal{L}^{\text{TUP}}_{Pd}$ by observing the following fact:

Proposition 5.2.4. *$\mathcal{L}^{\text{TUP}}_{Pd} \hookrightarrow S$ iff $\mathcal{L}_{Pd} \hookrightarrow S$, where $S = \mathcal{L}_{\mathtt{Q}}$ or $S = \mathcal{L}^{\text{fin}}_{\mathtt{Q}}$ for any* pure *$\mathtt{Q}$.*

Proof. The *if* direction is straightforward because $\mathcal{L}^{\text{TUP}}_{Pd} \hookrightarrow \mathcal{L}_{Pd}$.

The *only if* direction is to show that, given $\mathcal{L}^{\text{TUP}}_{Pd} \hookrightarrow S$, $\mathcal{L}_{Pd} \hookrightarrow S$. By Definition 14, for any *satisfiable* $D \in \mathcal{L}_{Pd}$, it follows that $D = \bigsqcap D_i$ where $D_i \in \mathcal{L}^{\text{TUP}}_{Pd}, 1 \leq i \leq n$. Considering $n \geq 2$ (when $n = 0$ and $n = 1$ the claim holds vacuously), by the assumption

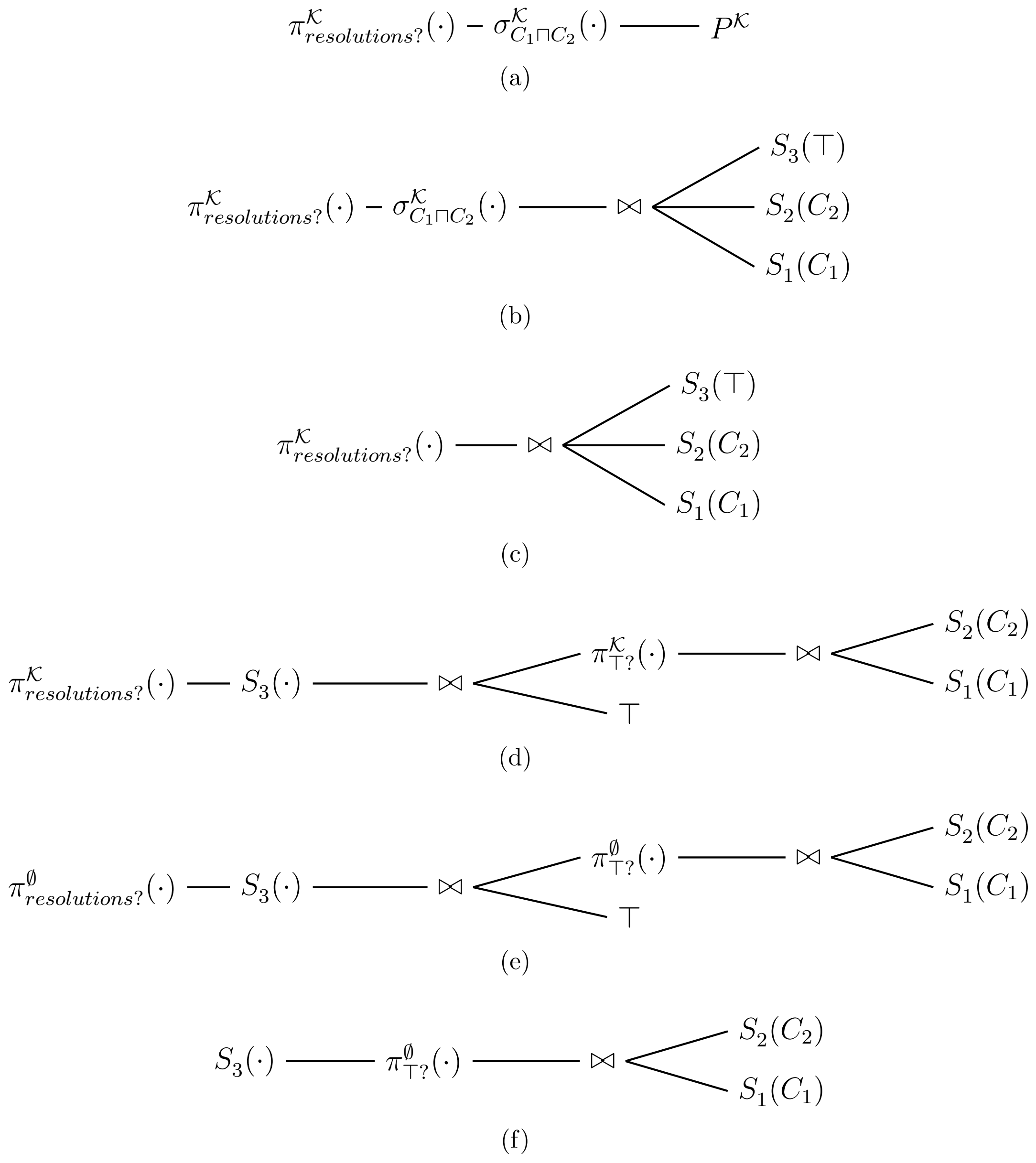

$\pi^{\mathcal{K}}_{resolutions?}(\cdot)$
$\sigma^{\mathcal{K}}_{C_1 \sqcap C_2}(\cdot)$
$P^{\mathcal{K}}$
(a)
$\bowtie$
$S_3(\top)$
$S_2(C_2)$
$S_1(C_1)$
(b)
(c)
$S_3(\cdot)$
$\pi^{\mathcal{K}}_{\top?}(\cdot)$
$\top$
(d)
$\pi^{\emptyset}_{resolutions?}(\cdot)$
$\pi^{\emptyset}_{\top?}(\cdot)$
(e)
(f)

Figure 5.3: Different query plans obtained by applying query rewriting rules to (5.11).

$\mathcal{L}^{\mathrm{TUP}}_{Pd} \hookrightarrow S$, for each satisfiable D_i, there is $E_i \in S$ such that $\models D_i \equiv E_i$. Consequently, $\models D \equiv \bigsqcap E_i, 1 \leq i \leq n$; furthermore, it follows from Corollary 5.2.1 that there exists $E \in S$ such that $\models E \equiv \bigsqcap E_i, 1 \leq i \leq n$, when S is either $\mathcal{L}_{\mathtt{Q}}$ or $\mathcal{L}^{\mathrm{fin}}_{\mathtt{Q}}$ for any *pure* $\mathtt{Q}$. Hence, $\models D \equiv E$ for some $E \in S$, where $S = \mathcal{L}_{\mathtt{Q}}$ or $S = \mathcal{L}^{\mathrm{fin}}_{\mathtt{Q}}$ for any *pure* $\mathtt{Q}$. □

A similar property can be established for determining $\mathcal{L}_{Pd} \hookrightarrow \mathcal{L}^{\mathrm{fin}}_{\mathtt{Q}'}$ when $Pd = Pd_1 \sqcap Pd_2$:

Proposition 5.2.5. *For any $Pd = Pd_1 \sqcap Pd_2$, $\mathcal{L}_{Pd} \hookrightarrow S$ iff $\mathcal{L}_{Pd_1} \hookrightarrow S$ and $\mathcal{L}_{Pd_2} \hookrightarrow S$, where $S = \mathcal{L}_{\mathtt{Q}}$ or $S = \mathcal{L}^{\mathrm{fin}}_{\mathtt{Q}}$ for any* pure *$\mathtt{Q}$.*

Proof. It suffices to prove that, by Lemma 5.2.4, $\mathcal{L}^{\mathrm{TUP}}_{Pd} \hookrightarrow S$ iff $\mathcal{L}^{\mathrm{TUP}}_{Pd_1} \hookrightarrow S$ and $\mathcal{L}^{\mathrm{TUP}}_{Pd_2} \hookrightarrow S$ for $S = \mathcal{L}_{\mathtt{Q}}$ or $S = \mathcal{L}^{\mathrm{fin}}_{\mathtt{Q}}$. The *only if* direction is easy to show. Observe that, by Definition 14, $\mathcal{L}^{\mathrm{TUP}}_{Pd_1} \hookrightarrow \mathcal{L}^{\mathrm{TUP}}_{Pd}$ because $\top \in \mathcal{L}^{\mathrm{TUP}}_{Pd_2}$ and $\forall C \in \mathcal{L}^{\mathrm{TUP}}_{Pd_1}, C \sqcap \top \in \mathcal{L}^{\mathrm{TUP}}_{Pd}$. For the same argument, $\mathcal{L}^{\mathrm{TUP}}_{Pd_2} \hookrightarrow \mathcal{L}^{\mathrm{TUP}}_{Pd}$.

The *if* direction. Assume for any $C_i \in \mathcal{L}^{\mathrm{TUP}}_{Pd_i}$, there is $D_i \in S$ such that $\models C_i \equiv D_i, i \in \{1, 2\}$. Because $S = \mathcal{L}_{\mathtt{Q}}$ or $S = \mathcal{L}^{\mathrm{fin}}_{\mathtt{Q}}$ for any *pure* $\mathtt{Q}$, by Corollary 5.2.1, there is $D \in S$ such that $\models D \equiv D_1 \sqcap D_2$. That is, for any $(C_1 \sqcap C_2) \in \mathcal{L}^{\mathrm{TUP}}_{Pd}$, there is such a $D \in S$ and $\models D \equiv C_1 \sqcap C_2$. □

The following proposition derives from Proposition 5.2.5 and the definition of $\mathcal{L}_{\mathtt{Q}_1 \bowtie \mathtt{Q}_2}$:

Proposition 5.2.6. *Let $\mathcal{L}_{\mathtt{Q}_1} = \mathcal{L}_{Pd_1}$ and $\mathcal{L}_{\mathtt{Q}_2} = \mathcal{L}_{Pd_2}$, then $\mathcal{L}_{Pd} \hookrightarrow \mathcal{L}_{\mathtt{Q}_1 \bowtie \mathtt{Q}_2}$, where $Pd = Pd_1 \sqcap Pd_2$.*

Given the above discussions, it is tempting to establish an efficient procedure via Definition 14 to determine $\mathcal{K}$-free rewriting. However, Lemma 5.2.7 simply excludes the possibility of such an efficient procedure using Definition 14.

Lemma 5.2.7. *The procedure in Definition 14 is* non-elementary*.*

Proof. Consider the case $\{C\} \hookrightarrow \mathcal{L}_{Pd}$, where $Pd = \exists R_1. \ldots \exists R_i.(f_1? \sqcap \cdots \sqcap f_j?)$. To determine if the relationship $\hookrightarrow$ holds, we need to compute $\mathcal{L}_{Pd}$. By Definition 14, we first compute $\mathcal{L}^{\mathrm{TUP}}_{f_k}$ for $1 \leq k \leq j$, and it is easy to see the size of $\mathcal{L}^{\mathrm{TUP}}_{f_k}$ is some constant, say c, using finite approximation. Again, $\mathcal{L}^{\mathrm{TUP}}_{(f_1? \sqcap \cdots \sqcap f_j?)}$ is obtained by a "cross product" fashion of $\mathcal{L}^{\mathrm{TUP}}_{f_k}$, which leads to a exponential blowup of size: c^j. Note the unusual requirement of computing $\mathcal{L}^{\mathrm{TUP}}_{\exists R.Pd_1}$: $\mathcal{L}_{Pd_i}$, instead of $\mathcal{L}^{\mathrm{TUP}}_{Pd_i}$, needs to be computed. Thus, the size of $\mathcal{L}_{\exists R_i.(f_1? \sqcap \cdots \sqcap f_j?)}$ is further augmented by the power-set expansion of $\mathcal{L}^{\mathrm{TUP}}_{(f_1? \sqcap \cdots \sqcap f_j?)}$, i.e., 2^{c^j}. Since each $R_k, 1 \leq k \leq i$ requires a power-set expansion, the size of $\mathcal{L}_{Pd}$ is i-EXPTIME. Because i is not bounded and, for each concept $D \in \mathcal{L}_{Pd}$, a logical equivalence check $\models C \equiv D$ is required, determining $\mathcal{K}$-free writing is thus non-elementary. □

In the following, an approximation procedure is outlined, in which determining $\mathcal{L}_{Pd_1} \hookrightarrow \mathcal{L}_{Pd_2}$ is reduced to structural comparison between Pd_1 and Pd_2. Observe that the procedure focuses on the representative language of the form $\mathcal{L}_{Pd}$, since other cases (see Definition 30) can be approximated by $\mathcal{L}_{Pd}$, such as the one shown by Proposition 5.2.6.

Definition 32. ***Decomposition Tree***. Let Pd be a projection description, the *decomposition tree* of Pd, denoted $\textsc{Deco}(Pd)$, is a triple (N, E, n), where N is a set of nodes, E a set of labelled edges in the form of (n, L, n'), where L is some label, and n the root of this tree. $\textsc{Deco}(Pd)$ is defined inductively as follow:

$$\begin{aligned}
&\textsc{Deco}(C?) = (\{n, n'\}, \{(n, C, n')\}, n)\\
&\textsc{Deco}(f?) = (\{n, n'\}, \{(n, f = *, n')\}, n)\\
&\textsc{Deco}(Pd_1 \sqcap Pd_2) = ((N_1 \cup N_2) \backslash \{n_2\}, (E_1 \cup E_2 \cup \{(n_1, L, n') \mid (n_2, L, n') \in E_2'\}) \backslash E_2', n_1)\\
&\qquad \text{where } \textsc{Deco}(Pd_i) = (N_i, E_i, n_i), i \in \{1, 2\} \text{ and } E_2' = \{(n_2, L, n') \mid (n_2, L, n') \in E_2\}\\
&\textsc{Deco}(\exists S.Pd_1) = (N_1 \cup \{n\}, E_1 \cup \{(n, S, n_1)\}, n) \text{ where } \textsc{Deco}(Pd_1) = (N_1, E_1, n_1)
\end{aligned}$$

□

With the definition of a decomposition tree, it is now possible to compare two projection descriptions, e.g., Pd_1 with Pd_2. The purpose of such a comparison is to find out if Pd_1 is less general than Pd_2, or, Pd_1 is *structurally included* in Pd_2, as defined below:

Definition 33. ***Structural Inclusion***. A projection Pd is structurally included in some decomposition tree $\textsc{DecoTree}$, denoted $\textsc{Stru}(Pd, \textsc{DecoTree})$, if any one of the following conditions holds, where $\textsc{DecoTree} = (N, E, n)$:

$$\begin{aligned}
&Pd = C?, \models C \equiv D, \text{ and there is } n' \in N \text{ such that } (n, D, n') \in E; \text{ or}\\
&Pd = C?, C = (f = k), \text{ and there is } n' \in N \text{ such that } (n, f = *, n') \in E; \text{ or}\\
&Pd = C?, C = C_1 \sqcap C_2, \textsc{Stru}(C_1?, \textsc{DecoTree}), \text{ and } \textsc{Stru}(C_2?, \textsc{DecoTree}); \text{ or}\\
&Pd = C?, C = \exists S.C_1, \text{ there is } n' \in N \text{ such that } (n, S, n') \in E,\\
&\qquad\qquad\qquad\qquad \text{and } \textsc{Stru}(C_1?, (N, E, n')); \text{ or}\\
&Pd = f? \text{ and there is } n' \in N \text{ such that } (n, f = *, n') \in E; \text{ or}\\
&Pd = Pd_1 \sqcap Pd_2, \textsc{Stru}(Pd_1, \textsc{DecoTree}), \text{ and } \textsc{Stru}(Pd_2, \textsc{DecoTree}); \text{ or}\\
&Pd = \exists S.Pd_1, \text{ there is } n' \in N \text{ such that } (n, S, n') \in E, \text{ and } \textsc{Stru}(Pd_1, (N, E, n'))
\end{aligned}$$

□

Note that $\{C\} \hookrightarrow \mathcal{L}_{Pd}$ in Theorem 5.2.2 can be reduced to $\mathcal{L}_{C?} \hookrightarrow \mathcal{L}_{Pd}$, hence it suffices to consider $\mathcal{L}_{Pd_1} \hookrightarrow \mathcal{L}_{Pd_2}$ for the $\mathcal{K}$-free rewriting conditions in Theorem 5.2.2. To determine if $\mathcal{L}_{Pd_1} \hookrightarrow \mathcal{L}_{Pd_2}$ holds, it is clearly more practical to approximate this task as structural inclusion tests:

Lemma 5.2.8. *$\mathcal{L}_{Pd_1} \hookrightarrow \mathcal{L}_{Pd_2}$ if* $\textsc{Stru}(Pd_1, \textsc{Deco}(Pd_2))$.

Proof. The proof proceeds by induction on Pd_1, assuming $\textsc{Deco}(Pd_2) = (N, E, n)$. (1) $Pd_1 = C?$. If $\textsc{Stru}(Pd_1, \textsc{Deco}(Pd_2))$, then (a) $\models C \equiv D$ and there is $n' \in N$ such that $(n, D, n') \in E$. By Definition 32, $(n, D, n') \in E$ implies $Pd_2 = D?$. Because $\models C \equiv D$, it follows that $\mathcal{L}_{Pd_1} \hookrightarrow \mathcal{L}_{Pd_2}$. (b) $C = (f = k)$ and there is $n' \in N$ such that $(n, f = *, n') \in E$. Again, by Definition 32, $Pd_2 = f?$, which implies $\mathcal{L}_{Pd_1} \hookrightarrow \mathcal{L}_{Pd_2}$ based on finite approximation (see Definition 31). (c) $C = C_1 \sqcap C_2$. This case reduces to the case $Pd = Pd_1 \sqcap Pd_2$ by observing that $\{C\} \hookrightarrow \mathcal{L}_{\mathtt{Q}}$ iff $\mathcal{L}_{C?} \hookrightarrow \mathcal{L}_{\mathtt{Q}}$ (since the concept $\top$ additionally introduced by $\mathcal{L}_{C?}$ is trivially included in $\mathcal{L}_{\mathtt{Q}}$ for every pure $\mathtt{Q}$) (d) $C = \exists S.C_1$, there is $n' \in N$ such that $(n, S, n') \in E$, and $\textsc{Stru}(C_1?, (N, E, n'))$. In this case, $Pd_2 = \exists S.Pd_3$ and $\mathcal{L}_{C_1?} \hookrightarrow \mathcal{L}_{Pd_3}$ by inductive hypotheses. By definition of $\mathcal{L}_{Pd}$, it follows $\mathcal{L}_{\exists S.C_1?} \hookrightarrow \mathcal{L}_{\exists S.Pd_3}$. (2) $Pd = f?$. This case is similar to $Pd = (f = k)?$. (3) $Pd = Pd_1 \sqcap Pd_2$. This follows directly from Proposition 5.2.5 and inductive hypotheses on Pd_1 and Pd_2. (4) $Pd = \exists S.Pd_1$. This case is similar to $Pd_1 = C?$, where $C = \exists S.C_1$. $\square$

The approximation procedure in Lemma 5.2.8 is sound and incomplete. The incompleteness is largely due to $C?$, in which C can be an arbitrary concept. For instance, $\mathcal{L}_{(A \sqcap B)?} \hookrightarrow \mathcal{L}_{(A \sqcup \neg B)? \sqcap B?}$ holds, yet it fails to hold by the approximation procedure. However, in practice user queries are more likely to be concerned with definite answers, which means disjunctions rarely occur in Pd.

5.3 Query Generation and Selection in CARE

Section 5.1 elaborates the rewriting rules that translate a given user query Q into various *query plans* that consist of a sequence of algebraic operators. To efficiently evaluate a query plan, a query engine must select the appropriate implementation of all the algebraic operators that participate in the query plan. This section discusses the implementation details on query generation and selection in the CARE assertion retrieval engine (see Section 4.4).

Recall that, similar to query compilation in relational databases, an algebraic operator can have multiple implementation (algorithms); for instance, the *join* operator in relational

DBMSs may be implemented as a *nested loops* join, a *sort-merge* join, or a *hash join*. Thus, a query plan may have more than one implementation that can be used for evaluation. Moreover, a user query can have at least one query plan due to query rewriting presented in the previous sections. With all the possibilities to answer a user query, a query engine must select a query plan that is likely to perform better than the rest. Such a phase is called *query planning* in this work. To elaborate query planning, Section 5.3.1 discusses how algebraic operators presented in Section 3.5.2 are implemented in CARE. Section 5.3.2 describes a straightforward strategy that derives from [Selinger et al., 1979] to select promising query plans in a plan space.

5.3.1 Implementing Operators in CARE

To execute a query plan, a straightforward way is to execute these operations in order and to store the results of each operation until it is needed by a subsequent operation, i.e., the *materialization* approach. Alternatively, multiple operations can be executed at once in such a way that no intermediate results are stored. In this *pipelining* approach, a single result (i.e., a concept assertion of the form $a : C$) produced by an operation is immediately passed to another operator. The latter approach is more appropriate in this work than the former since computing all the results of a single operator is potentially expensive. One disadvantage of pipelining operations is the inability to sort the results, however, it allows users to retrieve a certain number of answers and terminate the query answering process, in addition to saving memory for materializing results.

Pipelining algebraic operations can be achieved through an *iterator* that consists of a suite of primitive procedures, for example, `GetFirst` and `GetNext` in [Toman and Weddell, 2011], `Open`, `GetNext` and `Close` in [Molina et al., 1999]. In this work, two primitive functions are used: `Open` and `GetNext`. The former initiates the process of getting an answer, including, for example, setting the data structures required for later operations. The latter function returns the next result and adjusts the data structures and signals that indicates whether no more results are available, among others. In this work, an iterator is considered to have exhausted the resources once it returns `null`.

Pipelining a constant operation C is straightforward since this operation only produces one result $(\star : C)$. For the primary operation $P^{\mathcal{K}}$, `Open` maintains a list of concept assertions, one for each individual occurring in $\mathcal{K}$, and initializes the current cursor position to the beginning of the list. Then, `GetNext` returns the concept assertion which the cursor currently points to. Once the cursor goes beyond the range of the indices of the list, it issues a signal that there are no more results to be returned. The $\sigma^{\mathcal{K}}_{C}(\mathtt{Q})$ and $\pi^{\mathcal{K}}_{Pd}(\mathtt{Q})$ operations can be pipelined easily as well. In the case of selections, an iterator simply fetches one answer, say $a : D$, from $[\![\mathtt{Q}]\!]_{\mathcal{K}}^{\mathbb{SI}}$ and then decides if $\mathcal{K} \cup \{a : D\} \models a : C$. If so, the iterator returns it, otherwise, it fetches the next answer from Q. The projection case is similar, except that Algorithm 1 is used to compute an answer. The following discussions concentrate on how to pipeline a cache scan and a join operation.

Pipelining a Cache Scan

Recall that a cache scan $S_i(\mathtt{Q})$ exploits some cached query result, S_i, that is supported by a description index. Presumably, searching over a description index is efficient because the search condition generated by the iterator over Q is considered to be selective. In some circumstances, however, it may be preferable to perform a linear scan of a description index, for example, when the search condition is not selective. In this section, two approaches for iterating over a description index are discussed: one is search-based and the other is traversal-based.

Recall that a cache scan is defined by the plan $S_i(\mathtt{Q})$, where the cached query result S_i is defined by the user query $(D, Pd) : Od$. Note that a cached query result is always given an ordering description for efficient search. Because searching is performed over a description index, a cache scan invokes two iterators: one that iterates over Q and the other iterates over a description index. To ease the presentation, a description index is denoted T.

Search-based Iteration To search over a description index, the iterator that searches over a description index is implemented by Algorithms 8 and 9. Note that the search condition is given as a concept assertion $a : C$. A node in a description index contains the stored concept assertion (*data*), the links to the left and right children. The iterator maintains two sub-iterators (*leftIt* and *rightIt*), one over each subtree. The main idea here is to search over the left and right subtrees for a qualified answer and store them (possibly `null`) in an array of size three (*triple*). Also, if the root qualifies as an answer, it is stored in the array as well. Example 10 follows to illustrate the idea. Observe that it is possible to avoid searching part of a description index by comparing the concept assertion stored in the node (*root.data*) with the search condition ($a : C$), as shown in Example 10. Such comparison is performed by a helper function, Compare$(\cdot)$, which follows Definition 19 to determine the ordering between two concept assertions w.r.t. the Od for defining S_i. Specifically, it returns -1 if $\prec_{Od}$ $(a : C, root.data)$, 1 if $\prec_{Od}$ $(root.data, a : C)$, and 0 otherwise.

Once the array has solicited three answers, Algorithm 10 is used to choose the *right* one of the three that is to be returned as the final answer. Here, the definition of a *right* answer is dependent on the *ordering* constraint on the results. There are two options for specifying the ordering constraints: the results are to be ordered by the individual names (Ind) or by the null ordering (Un), in addition to satisfying the stipulated order, Od, in the definition of S_i. The first ordering requirement is useful if the results are to be pipelined into a join operation that adopts the *sort-merge* join. The function to determine the *right* answer is DetermineOrder($\cdot$), located on line 1 in Algorithm 10. The implementation of this function is left abstract because it is straightforward: the function returns the least indexed item in the array if the ordering requirement is Un, while it returns the smallest item in the array w.r.t. Ind otherwise. Once the right answer is decided, Algorithm 10 advances the appropriate sub-iterator to obtain the next answer. When *all* elements in the triple array are null, no more answers are available.

Algorithm 8: `Open`$(T, root, a : C)$ (Search-based)

```
1 $triple \leftarrow$ **null**
2 $leftIt \leftarrow$ **null**
3 $rightIt \leftarrow$ **null**
4 $T.root \leftarrow root$
```

Example 10. ***Search-based Iteration of a Cache Scan*** Consider a description index underlying some cached query results S_1 that consists of seven concept assertions stored w.r.t. $Od = age$: Un, as depicted in Figure 5.4. Suppose there is a cache scan $S_1(age = 20)$,

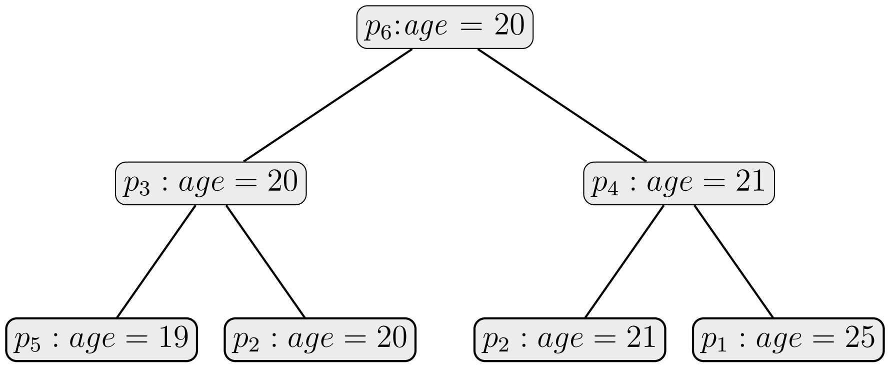


Figure 5.4: The description index used in Example 10.

i.e., searching over S_1 for all objects with age 20. Also, it is necessary that the qualified objects be returned in the order of their identifiers, i.e., Ind. Recall that the constant

Algorithm 9: `GetNext`() (Search-based)

```
1  $root \leftarrow T.root$
2  if $root$ = null then  return null
3  $dir \leftarrow$ Compare($a : C, root.data, Od$)
4  switch $dir$ do
5      case $-1$
6          $leftIt \leftarrow$ Open($T, root.rightChild, a : C$))
7          return $leftIt$.GetNext();
8      case 1
9          $rightIt \leftarrow$ Open($T, root.leftChild, a : C$))
10         return $rightIt$.GetNext();
11     case 0
12         if $root.data \models a : C$ then $triple[1] \leftarrow root$
           $leftIt \leftarrow$ Open($T, root.leftChild, a : C$)
13         $triple[0] \leftarrow this.leftIt$.GetNext()
14         $rightIt \leftarrow$ Open($T, root.rightChild, a : C$)
15         $triple[2] \leftarrow this.rightIt$.GetNext()
16         return ChooseNext($triple, this, a : C$)
```

operator in this cache scan returns a single, generalized concept assertion $\star : (age = 20)$. In the subsequent discussions, the term “search concept” in fact refers to this concept assertion. The search-based iteration idea works as follows.

First, the main iterator checks the root p_6. Because this node satisfies the search concept, it is a valid answer and is stored in a triple t_1, i.e, $t_1[1] = p_6$, where the positions $0, 1$, and 2 in a triple are reserved for the left, root, and right answers, respectively. To this end, it is not known if the subtrees of this node will return any other valid results. So, two iterators are created, one for each subtree.

Second, the newly created left iterator of p_6 checks the node p_3. Again, this node is a valid answer and is stored in another triple t_2, i.e., $t_2[1] = p_3$. Also, two more iterators must be created for this subtree. It is easy to check that the left sub-iterator returns no result since p_5 is not an answer, while the right iterator return the only answer p_2 from the triple $t_3 = (null, p_2, null)$. This answer, p_2, is from the right sub-iterator of p_3, so, $t_2[2] = p_2$ and finally, $t_2 = (null, p_3, p_2)$. Here, if no comparison is done, i.e., DetermineOrder($\cdot$) is not defined, then p_3 would be returned as the first answer, which violates the requirement to

Algorithm 10: ChooseNext(*triple*, *itr*)

```
1  $index \leftarrow$ DetermineOrder($triple, od$)
2  $ans \leftarrow$ null
3  switch $index$ do
4      case 0
5          $ans \leftarrow triple[0]$
6          $triple[0] \leftarrow itr.leftIt.$GetNext()
7      case 1
8          $ans \leftarrow triple[1]$
9          $triple[1] \leftarrow$ null
10     case 2
11         $ans \leftarrow triple[2]$
12         $triple[2] \leftarrow itr.rightIt.$GetNext()
13     return $ans.data$
```

return answers in the order of object identifiers. Hence, DetermineOrder($\cdot$) in this case must determine that p_2 is returned and t_2 is updated: $t_2 = (null, p_3, null)$. Consequently, t_1 is updated: $t_1[0] = p_2$, because p_2 is from the left sub-iterator of p_6.

Third, the right sub-iterator of p_6 checks the node p_4 and finds out the search concept is ordered before the concept assertion in this node. Hence, the iterator avoids searching the right subtree of p_4. Since the left subtree of p_4 returns no valid answers, this iterator returns no answers, which means $t_1[2] = null$.

Hence, $t_1 = (p_2, p_6, null)$. Furthermore, DetermineOrder($\cdot$) decides that p_2 is the first answer to be returned, i.e., $ans = (p_2)$ and $t_1 = (null, p_6, null)$. However, recall the left subtree of p_6 produces the triple $t_2 = (null, p_3, null)$, which now advances to the next valid answer p_3 and returns it to t_1, so, in the next call to GetNext($\cdot$), $t_1 = (p_3, p_6, null)$ and $t_2 = null$. Again, DetermineOrder($\cdot$) decides that p_3 in t_1 should be returned before p_6. Thus, the final answers are $ans = (p_2, p_3, p_6)$ in the required order. □

Traversal-based Iteration If the majority of nodes in a description index are to be returned as answers, it is more efficient to just traverse the index tree and then qualify the answers than searching over the tree, because the overhead of creating iterators for left and right subtrees is not negligible. In this work, a *right threaded* binary search tree is assumed for efficient transversal. In this right-threaded binary search tree, the right child of a node is either a real child node or a pointer, called a *thread*, to its *in-order* successor. In addition to the protocols defined previously for a description index, there is a boolean flag for each node to distinguish its right child node from a thread, denoted *threading*. To maintain the order defined in a cached query result, an iterator traverses the description index in an *in-order* fashion.

Algorithm 11: `Open`$(T, root, a : C)$ (Traversal-based)

```
1 $cur \leftarrow root$
2 $least \leftarrow$ false
3 $T.root \leftarrow root$
```

Algorithm 12: `GetNext`() (Traversal-based)

```
1 if $least =$ false then
2     if $cur \neq$ null then
3         while $cur.leftChild \neq$ null do
4             $cur \leftarrow cur.leftChild$
5     $least \leftarrow$ true
6     return $cur.data$;
7 else
8     if $cur.threading =$ false then
9         $cur \leftarrow cur.rightChild$
10        while $cur.leftChild \neq$ null do
11            $cur \leftarrow cur.leftChild$
12    else $cur \leftarrow cur.rightChild$ if $cur.data \models a : C$ then return $cur.data$ else
      return null
```

Algorithm 11 sets the current position of an iterator to the root of the description index and the flag for finding the least element (i.e., the first element in an in-order traversal) to false. Algorithm 12 is indeed a combination of `GetFirst`() and `GetNext`(). It returns the least element in the first iteration and traverses the description index, from the last returned position, in subsequent iterations.

Implementing Cache Scans The iterator functions for a cache scan can be defined based on an iterator for description indices. Algorithm 13 first opens an iterator over `Q` and produces a search condition. It then opens an iterator over the underlying description index. Algorithm 14 is also straightforward: it searches over the description index for the current search condition until none are returned, then, it advances the iterator for the next search condition and repeats the search.

There is, however, a caveat for cache scans of the form $S_i(C)$, i.e., the sub-iterator is over a constant operator. In this case, the sub-iterator only generates a single specialized concept assertion in the form of $(\star : C)$. Note that both Algorithms 9 and 12 take as an argument a *genuine* concept assertion. In this case, $(\star : C)$ needs to be translated into $(a : C)$, where $a : D$ is the concept assertion occurring in the node being searched over.

Algorithm 13: `Open`$(S_i, \mathtt{Q})$ (Cache Scans)

```
1 $subIt \leftarrow$ Open(Q)
2 $ca \leftarrow subIt.$GetNext()
3 $T \leftarrow S_i.\textsc{GetDescriptionIndex}()$
4 $indexIt \leftarrow$ Open$(T, T.root, ca)$
```

Algorithm 14: `GetNext`() (Cache Scans)

```
1 if $ca$ = null then  return null $ans \leftarrow indexIt.$GetNext$(T, T.root, ca)$
2 if $ans$ = null then
3     $ca \leftarrow subIt.$GetNext()
4     if $ca$ = null then return null else
5         $indexIt \leftarrow$ Open$(T, T.root, ca)$
6         return GetNext();
```

Notice that Lines 4 and 5 in Algorithm 13 and 14, respectively, needs to choose the proper iteration method over a description index. As briefly discussed earlier, when the search condition is selective, it is advantageous to use search-based iteration. Beyond selectivity, other factors may also affect the choice between these two methods. In what follows, heuristics for selecting a more promising iteration method in CARE are summarized. Consider a cache scan $S_1(C)$, where the cached query result S_1 is defined by the query $(C_1, Pd_1) : Od_1$.

First, the descriptive sufficiency (see Definition 20) of concept assertions w.r.t. Od_1 in a cached query result affects the search performance. For a node in which the concept assertion is insufficiently descriptive, searching over that node has to generate two sub-iterators, one for each subtree. If there are a large number of such nodes in a description index, the search performance will be severely degraded. Hence, for a description index of which the proportion of nodes with insufficiently descriptive concept assertions is above some threshold, it is more favourable to apply *traversal-based* iterations.

Second, the selectivity of the search condition and the concept assertions in the description index is an important factor in deciding a proper iteration method. Considering $C = (f < k)$, the selectivity of this constant operator depends on the range of all the values occurring in the description index. Statistical information about the concept assertions in a description index needs to be taken into consideration when deciding which iteration method is more efficient.

Third, Od_1 is also a factor for deciding a proper iteration method. Particularly, if $Od_1 = f : Od_2$ and the search condition is not about the same feature, e.g., $(g = k)$, then *traversal-based* iterations are preferred because the search condition is incompatible with the major sort of Od_1.

Last but not least, if the search condition is trivially true, then all concept assertions in a description index qualify as answers, for example, when the constant operator $C = \top$ or it holds that $\models C_1 \sqsubseteq C$. In this case, it is more efficient to use *traversal-based* iterations.

Pipelining a Join

A join operation $\mathtt{Q}_1 \bowtie \mathtt{Q}_2$ is analogous to a natural join in relational algebra, viewing individual names in concept assertions as the common attribute. There are several ways to implement a join operation and this section presents two common iterative join methods: *nested loops* join and *sort-merge* join.

Algorithm 15: $\mathtt{Open}(\mathtt{Q}_1, \mathtt{Q}_2)$ (Join)

```
1 $leftIt \leftarrow \mathtt{Open}(\mathtt{Q}_1)$
2 $rightIt \leftarrow \mathtt{Open}(\mathtt{Q}_2)$
3 $ca_1 \leftarrow leftIt.\mathtt{GetNext}()$
```

Algorithm 15 opens two iterators, one for each sub query plans, and fetches one result from the left sub query plan. Note that both nested loops and sort-merge joins use this algorithm for $\mathtt{Open}(\cdot)$, though sir-merge join assumes that both sub iterators return results in order (w.r.t. Ind). Algorithm 16 shows how to process a join operation using nested loops, while Algorithm 17 is for sort-merge join. Observe that both algorithms assume that a concept assertion consists of two components: the individual name (*ind*) and the

Algorithm 16: `GetNext`() (Nested loops)

```
if $ca_1$ = null then  return null $ca_2 \leftarrow$ $rightIt$.GetNext()
if $ca_2$ = null then
    $ca_1 \leftarrow$ $leftIt$.GetNext()
    if $ca_1$ = null then  return null $rightIt \leftarrow$ Open($Q_2$)
    $ca_2 \leftarrow$ $rightIt$.GetNext()
if $ca_1.ind = ca_2.ind$ then
    $a \leftarrow ca_1.ind$
    $C_1 \leftarrow ca_1.cp$
    $C_2 \leftarrow ca_2.cp$
    return $a : C_1 \sqcap C_2$
else return null
```

Algorithm 17: `GetNext`() (Sort-merge)

```
if $ca_1$ = null then  return null $ca_2 \leftarrow$ $rightIt$.GetNext()
while $ca_1 \neq$ null and $ca_2 \neq$ null do
    $cmp \leftarrow$ Compare($ca_1, ca_2$, Ind)
    switch $cmp$ do
        case $-1$
            $ca_1 \leftarrow$ $leftIt$.GetNext()
        case $1$
            $ca_2 \leftarrow$ $rightIt$.GetNext()
        case $0$
            $a \leftarrow ca_1.ind$
            $C_1 \leftarrow ca_1.cp$
            $C_2 \leftarrow ca_2.cp$
            return $a : C_1 \sqcap C_2$
        else return null
```

concept description (cp). Finally, joining constant operators should follow what is described by the semantics (see Table 3.1). The only modification in Algorithms 16 and 17 is to let Compare($\cdot$) return 0 whenever one of the concept assertion is of the form ($\star : C$).

5.3.2 Plan Selection in CARE

A user query can derive several query plans that differ in complexity of query answering by applying query rewriting rules. These plans form a plan space in which a query optimizer needs to choose the most *promising* plan using certain heuristics. Similar to cost-based plan selection in relational databases, the heuristics used during plan selection are based on cost estimation, which assigns a cost to each query plan in the plan space. In addition, a search algorithm is required to search through the plan space. This section presents a simple yet effective cost model implemented in CARE and the search algorithm used for query planning, which derives from [Selinger et al., 1979].

Cost Estimation

Cost estimation of resources has been extensively used in relational DBMSs to guide the search for query plans. Since this work concerns in memory data, the resources to be considered exclude IO cost, communication cost, among others. In particular, the resources are mainly about CPU time. It is clear that query optimization is only as accurate as the cost estimates, thus, this section addresses how statistical data can be collected, how to determine the statistical summary of the output data of an operation, and the cost estimates of executing an operation.

Concept Selectivity The selectivity of concepts is of paramount importance, as they are used as selection conditions in a selection operator, search conditions in cache scans, among others. Considering any $\mathcal{ALCI}(\mathbb{D})$ concept, the selectivity depends on the selectivity of the concept names, role names, and features occurring in it, as well as the propagation of selectivity through connectives. Note that selectivity is estimated w.r.t. some $\mathcal{K}$. When the context is clear, the knowledge base is omitted. The selectivity estimate now follows.

One of the differences between databases and knowledge bases is the structure of data. For a DL knowledge base, the ABox data is not structured as columns. However, features are analogous to attributes in relations. In many large information systems, histograms are often used to collect information on the distribution of the data in some column. Analogously, histograms can be used to record the data of a particular feature. A coarse-grained approach can simply record the range of values for this feature. Let $\texttt{sizeOf}(f)$ denote the number of values that a feature f takes on in the ABox. In addition, not every individual will use the feature f, so, $\texttt{sizeOf}(f = *)$ is used to denote the number of individuals that mentions f.

For each concept name A occurring in a knowledge base, the number of individuals that are *explicitly* declared to belong to A can be collected. Intuitively, the size of a concept A can indicate how selective A is over the ABox data. The size of a named role S can be estimated similarly. The number of told role assertions $S(a, b)$ in a knowledge base indicates the number of individuals associated with this role. A more refined estimate can combine the aforementioned estimate for concepts such that the size of S can be stored in a histogram to estimate how frequently S is associated with a particular concept A, as well as the promiscuity of S. When estimating the size of A and S, it only makes sense to record the explicitly told information, e.g., an individual that is told to be A, because otherwise a significant number of general reasoning requested are needed, which makes cost estimation infeasible. Let $\texttt{sizeOf}(A)$ and $\texttt{sizeOf}(S)$ denote the estimated size of a primitive concept and role, respectively.

It is possible to calculate the selectivity of a general $\mathcal{ALCI}(\mathbb{D})$ concept C by the following definition:

Definition 34. ***Concept Selectivity***. The selectivity of a $\mathcal{ALCI}(\mathbb{D})$ concept C, denoted $\texttt{sel}(C) \in [0, 1]$, is computed as follows:

$$\begin{aligned}
&\texttt{sel}(A) = \texttt{sizeOf}(A)/\texttt{sizeOf}(\top)\\
&\texttt{sel}(f = k) = (\texttt{sizeOf}(f = *)/\texttt{sizeOf}(\top)) \times 1/\texttt{sizeOf}(f)\\
&\texttt{sel}(f < k) = c_r\\
&\texttt{sel}(C_1 \sqcap C_2) = \texttt{sel}(C_1) \times \texttt{sel}(C_2)\\
&\texttt{sel}(\exists S.C_1) = (\texttt{sizeOf}(R)/\texttt{sizeOf}(\top)) \times \texttt{sel}(C_1)\\
&\texttt{sel}(\neg C_1) = 1 - \texttt{sel}(C_1)
\end{aligned}$$

where $c_r \leq 1$ is a constant that determines the selectivity of a range query. □

Note that the selectivity of $\exists S.C_1$ is calculated based on two factors: how frequently S is associated with other individuals and what is the likelihood of the associated individuals to be an instance of C_1.

Operator Size Estimation When an operator is evaluated, the size of output data is a crucial aspect of execution time. For some cached query result S_i, $\texttt{sizeOf}(S_i)$ denotes the number of concept assertions stored in it.

Definition 35. ***Operator Size***. The size of an algebraic operator $\texttt{Q}$, denoted $\texttt{sizeOf}(\texttt{Q})$, can be estimated as follows:

$$\texttt{sizeOf}(C) = 1$$

$$\begin{aligned}
&\texttt{sizeOf}(P^{\mathcal{K}}) = \texttt{sizeOf}(\top)\\
&\texttt{sizeOf}(S_i(\texttt{Q}_1)) = \texttt{sel}(C) \times \texttt{sizeOf}(S_i) \text{ if } \texttt{Q}_1 = C, \text{ or } c_s \times \texttt{sizeOf}(\texttt{Q}_1) \text{ otherwise}\\
&\texttt{sizeOf}(\sigma^{\mathcal{K}}_C(\texttt{Q}_1)) = \texttt{sel}(C) \times \texttt{sizeOf}(\texttt{Q}_1)\\
&\texttt{sizeOf}(\pi^{\mathcal{K}}_{Pd}(\texttt{Q}_1)) = \texttt{sizeOf}(\texttt{Q}_1)\\
&\texttt{sizeOf}(\texttt{Q}_1 \bowtie \texttt{Q}_2) = \texttt{min}(\texttt{sizeOf}(\texttt{Q}_1), \texttt{sizeOf}(\texttt{Q}_2))
\end{aligned}$$

where $c_s < 1$ is a constant that estimates roughly, given a concept assertion $a : D$ produced by the subquery $\texttt{Q}_1$, the likelihood to find a concept assertion in S_i. □

Note that c_s is guaranteed to be less than 100% because the individual a is assumed to match at most one concept assertion in S_i. Also note that refinement can be done for size estimation, for example, $\texttt{sel}(C)$ over S_i should be different from that over the original $\mathcal{K}$, because the cached query may significantly affect the original selectivity of C. The size of a join operation also needs to be estimated more carefully: when either $\texttt{Q}_1$ or $\texttt{Q}_2$ (but not both) are a constant operator, the function $\texttt{min}(\cdot)$ should really be $\texttt{max}(\cdot)$.

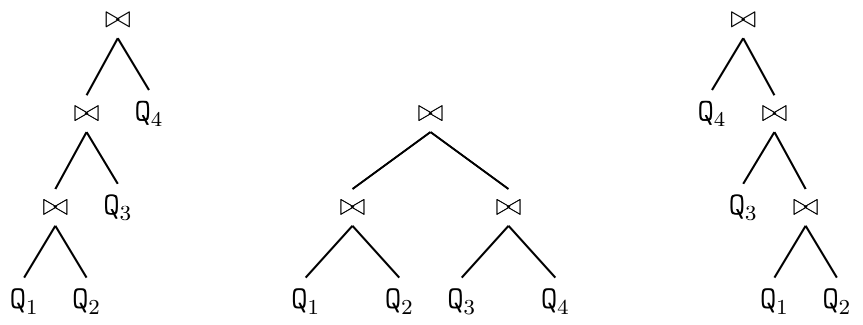
Cost of Executing a Query Plan The cost associated with the execution of an algebraic plan $\texttt{Q}$ is dependent on the size of the output data of the subqueries and the characteristics of the query plan itself. Therefore, it is reasonable to define a constant c for each of the six types of operations. Then, the cost of a query plan $\texttt{Q}$, denoted $\texttt{cost}(\texttt{Q})$, can be simply defined to be the product of c and the size of the subquery. Although the actual values of these constants do not matter, these values should reflect the cost of evaluating that particular type of operation. For instance, evaluating $\sigma^{\mathcal{K}}_C(\texttt{Q})$ ($\pi^{\mathcal{K}}_{Pd}(\texttt{Q})$, respectively) is, in practice, more expensive than evaluating $\sigma^{\emptyset}_C(\texttt{Q})$ ($\pi^{\emptyset}_{Pd}(\texttt{Q})$, respectively), because the former require general reasoning w.r.t. some knowledge base $\mathcal{K}$. Note that this does not imply reasoning about tautologies is easy: the complexity of deciding $\models C$ for a $\mathcal{ALC}$ concept C is already PSPACE-COMPLETE. Also, the primary operator, $P^{\mathcal{K}}$, provides the default access to individuals in $\mathcal{K}$, but it usually does not presume any inference for a consistent $\mathcal{K}$: the individuals are usually indexed during the knowledge base load phase.

What is more interesting is the cost of projection descriptions in projection operations. Recall the procedures in Section 3.4: computing projections requires a significant number of instance checking, which inevitably incurs a high computational overhead. To estimate the cost of executing a projection operation more reasonably, it is necessary to estimate the complexity involved in a projection description. For example, computing $\pi^{\mathcal{K}}_{f? \sqcap A?}(\texttt{Q})$ is likely more costly than computing $\pi^{\mathcal{K}}_{\top ?}(\texttt{Q})$. Appropriate measures can be made to account for the complexity of Pd. A straightforward approach is to measure the size of Pd and assign a constant for each base case of Pd, e.g., $f?$, $\top?$, or $A?$.

Join Order

Section 5.1 discusses several query rewriting techniques; in particular, rules that transform algebraic operators that require reasoning w.r.t. $\mathcal{K}$ into ones that do not are most interesting, because of the high complexity associated with general DL reasoning. However, in general, transformations do not necessarily result in cost-effective query plans, hence, a search algorithm must be able to avoid query plans that can incur higher cost for query execution.

Since the plan space is large, it is necessary to limit the plan space for efficient plan selection. In particular, join operations are among the first to be addressed. Because join operations are commutative and associative, they can be arranged in a number of ways, each representing a query plan. Consider the join of four query plans, $\mathtt{Q}_i, 1 \leq i \leq 4$. Figure 5.5 lists three possible join orders, which are logically equivalent. If a query optimizer wants to generate all possible join orders and compute the cost of each order, it is computationally intensive when the number of joins is large. In relational systems, linear join sequences, e.g., 5.5a and 5.5c, are more often used than bushy ones [Chaudhuri, 1998], e.g., 5.5b. The following discussions thus focus on *left deep* join trees.



(a) left deep (b) bushy (c) right deep

Figure 5.5: Different types of join trees.

The join optimization step is in resemblance to the exemplary System R optimization technique. Recall that System R enumerate joins in a bottom-up fashion [Selinger et al., 1979]. Specifically, it employs *dynamic programming* to obtain an optimal plan for k joins: it first finds the optimal plans for $k-1$ joins and then join those plans with the k-th relation. For the base case, i.e., $k = 1$, simply return the operator itself.

For example, to find the join orders for the four subqueries in Figure 5.5, it suffices to consider the best plans for the following joins: $\{\mathtt{Q}_1, \mathtt{Q}_2, \mathtt{Q}_3\} \bowtie \mathtt{Q}_4$, $\{\mathtt{Q}_1, \mathtt{Q}_2, \mathtt{Q}_4\} \bowtie \mathtt{Q}_3$,

$\{\mathtt{Q}_1, \mathtt{Q}_3, \mathtt{Q}_4\} \bowtie \mathtt{Q}_2$, and $\{\mathtt{Q}_2, \mathtt{Q}_3, \mathtt{Q}_4\} \bowtie \mathtt{Q}_1$. Observe that a naive approach that enumerates all permutations of k joins produces $\mathrm{O}(n!)$ query plans, while the above dynamic programming approach reduces the number of plans to $\mathrm{O}(n2^{n-1})$ [Chaudhuri, 1998].

Plan Space and Searching

Since most of the rewriting rules presented in Section 5.1 are equivalent transformations, they can result in a large number of equivalent plans. In addition, some operators can have more than one implementation. All these observations imply that the plan space for a user query is immense. Due to the limited resources allowed for query optimization, it is thus impossible to consider every query plan in the plan space. As stated earlier in this section, only left deep plans for join operations are considered.

To exploit the estimated cost of a query plan, dynamic programming can be used as the search algorithm over the plan space. Given a user query (C, Pd), a set of cached query results CACHE, and a knowledge base $\mathcal{K}$, the goal of the search algorithm is to find a query plan that has the lowest cost based on the cost model defined above. Also, APPLYRULE$(\cdot)$ denotes a function that maps the argument plan to another plan via applications of rewriting rules given in Section 5.1; JOINORDER(S) denotes the dynamic programming approach described above to find the best join order for a set of operators.

1 Generate the default plan $\mathtt{Q} = \pi^{\mathcal{K}}_{Pd}(\sigma^{\mathcal{K}}_{C}(P^{\mathcal{K}}))$. Let

$$\mathtt{Q}_s = \{\pi^{\mathcal{K}}_{Pd}(\sigma^{\mathcal{K}}_{C}(\mathtt{Q}')) \mid \mathtt{Q}' = \textsc{joinOrder}(\textsc{Idx}_i), \textsc{Idx}_i \subseteq \textsc{Cache}, \text{ and } \textsc{applyRule}(\mathtt{Q}) = \pi^{\mathcal{K}}_{Pd}(\sigma^{\mathcal{K}}_{C}(\mathtt{Q}'))\}.$$

Compute $\mathtt{cost}(\mathtt{Q}_i)$ for each $\mathtt{Q}_i \in \mathtt{Q}_s \cup \{\mathtt{Q}\}$. $\textsc{Plan} = \{\mathtt{Q}^1\}$ if $\mathtt{cost}(\mathtt{Q}_i)$ is the lowest or $\mathtt{Q}_i$ is an interesting plan, where $\mathtt{Q}_i = \pi^{\mathcal{K}}_{Pd}(\sigma^{\mathcal{K}}_{C}(\mathtt{Q}^1))$.

2 Let $\mathtt{Q}_s = \{\pi^{\mathcal{K}}_{Pd}(\mathtt{Q}') \mid \textsc{applyRule}(\pi^{\mathcal{K}}_{Pd}(\sigma^{\mathcal{K}}_{C}(\mathtt{Q}^1))) = \pi^{\mathcal{K}}_{Pd}(\mathtt{Q}'), \mathtt{Q}^1 \in \textsc{Plan}\}$. Compute $\mathtt{cost}(\mathtt{Q}_i)$ for each $\mathtt{Q}_i \in \mathtt{Q}_s$. $\textsc{Plan} = \{\mathtt{Q}^2\}$ if $\mathtt{cost}(\mathtt{Q}_i)$ is the lowest or $\mathtt{Q}_i$ is an interesting query plan, where $\mathtt{Q}_i = \pi^{\mathcal{K}}_{Pd}(\mathtt{Q}^2)$.

3 Let $\mathtt{Q}_s = \{\mathtt{Q}^3 \mid \textsc{applyRule}(\pi^{\mathcal{K}}_{Pd}(\mathtt{Q}^2)) = \mathtt{Q}^3, \mathtt{Q}^2 \in \textsc{Plan}\}$. Compute $\mathtt{cost}(\mathtt{Q}_i)$ for each $\mathtt{Q}_i \in \mathtt{Q}_s$. Output $\mathtt{Q}_i$ where $\mathtt{cost}(\mathtt{Q}_i)$ is the lowest.

Figure 5.6: A procedure for plan selection.

Figure 5.6 lists the steps to produce the most promising plan w.r.t. a given cost model. The general idea is as follows. First, the default query plan is generated, which serves as the seed plan in the plan space. The plan selection algorithm proceeds in a bottom-up fashion. Because the default plan consists of three operators, the algorithm starts from the bottom operator, namely, the primary operator $P^{\mathcal{K}}$. At this stage, the objective is to introduce cache query results such that the primary operator can be substituted by a set of cache scans (see Lemma 5.1.1). Also, if there are any rewriting rules applicable, then more query plans can be obtained. For all the plans generated during this stage, the algorithm computes the cost of each plan and only keeps the plan with the cheapest cost, as well as the plans that have a relatively high cost but are considered *interesting*.

The idea of *interesting* plans is similar to the *interesting order* presented in [Selinger et al., 1979], which has later been generalized as *physical properties* in [Graefe and DeWitt, 1987]. Specifically, such plans (more rigorously, operators) have certain features that can impact the cost of subsequent operations. One type of interesting plan is the cache scans that can benefit a subsequent selection operation. For instance, a set of cache scans may result in the removal of a selection operation (Lemma 5.1.2). In these cases, the joins of cache scans may appear to be expensive operations; however, if the cost of a subsequent selection operation can be significantly reduced, a final plan using these cache scans may result in a lower cost. Another class of interesting cache scans may benefit subsequent projection operations. For instance, the Pd that defines the cached query result in a cache scan may be the same as a subsequent projection operation. In this situation, the subsequent projection operations can be completely removed. More generally, a set of cache scans may lead to $\mathcal{K}$-free reasoning of a selection or projection operation. A more thorough discussion has been given in Section 5.2.

Back to the plan selection algorithm, stage 1 produces several operators that can replace $P^{\mathcal{K}}$. Stage 2 then works on $\sigma^{\mathcal{K}}_{C}(\cdot)$ in the same way as in stage 1: for each operator produced in the previous stage, the algorithm forms a substitute plan, applies the rewriting rules, computes the cost of all plans, and keeps the cheapest and interesting plans for the next stage. Note that interesting plans are propagated through stages and may be discarded once they become *uninteresting* in some stage. Stage 3 follows the same pattern, except that the final output is a single query plan, which is considered to be the most promising plan and is passed to a plan interpreter for execution.

On Parameterized User Queries

When a user issues a query, the query processor compiles the query by producing a query plan that can be executed over the knowledge base. The query optimization process is nevertheless expensive. Recall Chapter 1 mentions that a user query in the form of (C, Pd) can involve parameters to speedup query optimization. Unlike SQL queries, in which the predicates are features, an object query over a knowledge base can include an arbitrary concept in certain DL dialect. Therefore we only consider a user query in which the values of a feature can be parameterized.

A parameterized query usually represents a class of queries that are similar to one another. This generalization simplifies query optimization because one could use the same physical plan to execute all the queries in this class and query optimization needs to be performed only once for the whole class. Specifically, a user can submit a query with values of features missing, that is, the values are not provided until runtime. For instance, the concept part of the query presented in (1.1) can be characterized by the following parameterized query

$$(Digital\_SLR \sqcap \exists dealerPrice.(price < ?) \sqcap (releaseDate > ?), Pd)$$

where "?" stands for parameters and will be initialized during plan execution phase.

To pick the most promising plan for a parameterized query, the system can optimize the user query with choosing values for the parameters and use the chosen plan for all subsequent queries in this class. The suggested values for parameters can be chosen based on a statistical summary of query logs and the knowledge base, query optimization will use the suggested value to generate a plan. More complicated strategies can also be used, such as those used in commercial DBMSs. For instance, Goldstein et al. [2006] shows how MS SQL Server can produce a dynamic query plan that embeds more than one plan options within the plan. During plan execution, the set of options can lead to a particular plan option, depending on the constants being provided. This approach has the advantage of applying a universal plan to a class of queries and thus avoiding the bias towards a particular set of values for the parameters. The tradeoff here is the computational cost to produce such a dynamic plan.

5.4 An Empirical Evaluation

An empirical evaluation was carried out using CARE (see Section 4.4). The evaluation was over the DPC knowledge bases described in Section 4.4.1, i.e., DPC1 and DPC2. In addition, three cached query results were selected from the cache store, as given below:

$$\begin{aligned}
S_1 &= (\mathit{Digital\_SLR}, \mathit{user\_review}?) :: \mathit{user\_review} : \mathrm{Ind}\\
S_2 &= (\mathit{Digital\_Camera}, \mathit{release\_date}? \sqcap \mathit{x\text{-}dim}? \sqcap \mathit{y\text{-}dim}?) :: \mathit{release\_date} : \mathrm{Ind}\\
S_3 &= (\top, \exists \mathit{hasManu}.(\mathit{manu\_name}? \sqcap \exists \mathit{locatedIn}.\mathit{Europe\_Country}?)) :: \mathrm{Ind}
\end{aligned}$$

Four user queries were tested for query optimization, as given in Table 5.1. Also, the most cost-effective query plan selected by CARE for each test query is given in Table 5.1 (the query plans selected over DPC2 were the same as over DPC1). Recall that, for any

Q_1	$(\mathit{Digital\_SLR} \sqcap (\mathit{user\_review} = 4.00), \top?)$
$\mathtt{Q}_1$	$\pi^{\emptyset}_{\top?}(S_1(\mathit{user\_review} = 4.00))$
Q_2	$(\mathit{Digital\_Camera} \sqcap \neg(\mathit{release\_date} < 20100101), \mathit{release\_date}?)$
$\mathtt{Q}_2$	$\pi^{\emptyset}_{\mathit{release\_date}?}(\sigma^{\mathcal{K}}_{\mathit{Compact\_Camera} \sqcap \neg(\mathit{release\_date} < 20100101)}(S_2(\top)))$
Q_3	$(\exists \mathit{hasManu}.\exists \mathit{locatedIn}.\mathit{Europe\_Country}, \mathit{x\text{-}dim}? \sqcap \mathit{y\text{-}dim}?)$
$\mathtt{Q}_3$	$\pi^{\mathcal{K}}_{\mathit{x\text{-}dim}? \sqcap \mathit{y\text{-}dim}?}(S_3(\exists \mathit{hasManu}.\exists \mathit{locatedIn}.\mathit{Europe\_Country}))$
Q_4	$(\mathit{Digital\_SLR} \sqcap (\mathit{release\_date} = 20020222) \sqcap (\mathit{user\_review} = 4.50), \mathit{x\text{-}dim}? \sqcap \mathit{y\text{-}dim}?)$
$\mathtt{Q}_4$	$\pi^{\emptyset}_{\mathit{x\text{-}dim}? \sqcap \mathit{y\text{-}dim}?}(\sigma^{\mathcal{K}}_{C}(S_2(\mathit{release\_date} = 20020222)))$

Table 5.1: Test queries and the corresponding query plans chosen by CARE. Q_i and $\mathtt{Q}_i$ refer to the original query and the query plan, respectively. $C = \mathit{Digital\_SLR} \sqcap (\mathit{release\_date} = 20020222) \sqcap (\mathit{user\_review} = 4.50)$.

user query (C, Pd), there is a default plan that can be evaluated over some knowledge base $\mathcal{K}$, i.e., $\pi^{\mathcal{K}}_{Pd}(\sigma^{\mathcal{K}}_{C}(P^{\mathcal{K}}))$. It can be seen from Table 5.1 that CARE seemed to prefer plans that consist of $\mathcal{K}$-free operators and that have some of the operators completely removed from the default plan. The selection of such plans by CARE is congruous with the intuition that selection and projection operations are expensive to evaluate over an underlying $\mathcal{K}$.

Figure 5.7 delineates the execution times of the default plans and the optimized plans, as well as the time spent in query optimization for each test query. For the point query Q_1 the selected plan $\mathtt{Q}_1$ uses a cached scan and a $\mathcal{K}$-free projection operation and is free of selection operations. All these optimizations makes $\mathtt{Q}_1$ extremely efficient to evaluate: it is a purely structural plan. Q_2 is a range query, and $\mathtt{Q}_2$ scans the cached query result S_2

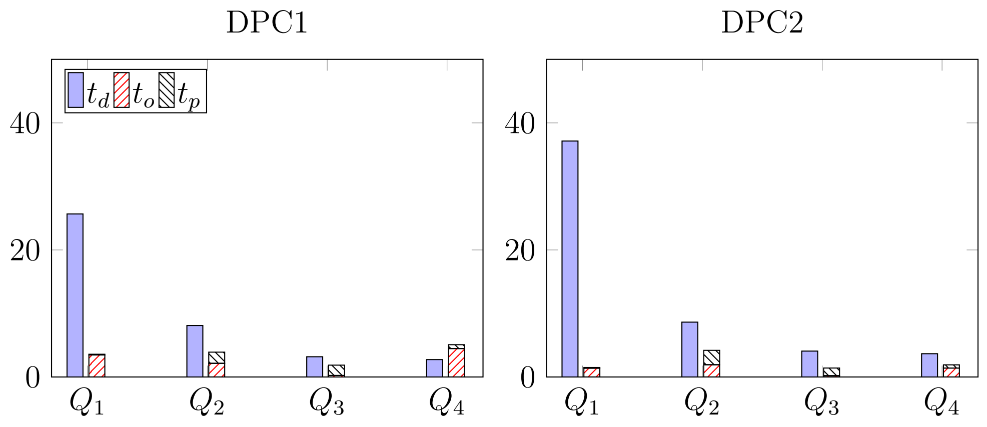


Figure 5.7: Query optimization and query execution times in seconds. t_d, t_o, and t_p refer to the time to execute the default plan of a user query Q_i, the time to perform query optimization, and the time to execute the corresponding query plan $\mathtt{Q}_i$ selected by CARE, respectively.

on the feature *release_date*. Because of this cached query result, $\mathtt{Q}_2$ is able to obtain a $\mathcal{K}$-free projection operation, which significantly reduced the execution time compared to the default plan. For Q_3 the chosen plan $\mathtt{Q}_3$ involves a cache scan on the cached query result S_3 with the selection condition, which enables the dismissal of the selection operation. Notice that $\mathtt{Q}_3$ still uses projection operations to get the x and y dimensions, which could have been avoided by another cache scan over S_2. In fact, CARE selected $\mathtt{Q}_3$ because a join of two cache scans using nested loops (note that merge join is not available in this case because S_2 is not ordered by Ind alone) is also costly, which could jeopardize the benefits of such a join. In the case of $\mathtt{Q}_3$, the system decided that performing projections over a small set of concept assertions is less costly than the previous join of two cache scans. The selection of $\mathtt{Q}_4$ is similar to that of $\mathtt{Q}_3$ because an alternative could be to join the scan over S_2 with another cache scan $S_1(\mathit{user\_review} = 4.50)$. Such a join would eliminate the selection operation that exists in $\mathtt{Q}_4$, as well as the projection operation. However, for the same reasons discussed in the case $\mathtt{Q}_3$, the chosen plan avoids such a join because the cache scan over S_2 only returns a few answers and performing selection or projection operations in this situation may be more preferable. Of course, the decisions made by CARE to choose $\mathtt{Q}_3$ and $\mathtt{Q}_4$ could be seriously wrong in situations where the underlying $\mathcal{K}$ is so complicated that reasoning tasks should be kept to a minimum. To obtain more reasonable plans in these cases, a more refined cost model than the one presented in this chapter is required. In particular, the "hardness" of an inference task w.r.t. $\mathcal{K}$ needs to be considered for the given $\mathcal{K}$, while the current cost model weighs all inference tasks w.r.t. different knowledge bases the same way. Nevertheless, there is no known, accurate approach to measure the "hardness" of any given knowledge base.

As shown in Figure 5.7, all the query plans selected by CARE are far more efficient than the default plans of the original test queries. Furthermore, query optimization in all cases only took a few seconds. Observe that the query optimization time for Q_4 over DPC1 outweighed the execution time of the default plan. Such a situation could arise if executing the default plan takes little time, however, when a query such as Q_4 is executed repeatedly (in this case, running Q_4 twice suffices), the optimized query plan enables more efficient query execution overall.

This chapter presents a query optimization framework for assertion queries over knowledge bases. In particular, $\mathcal{K}$-free rewritings are discussed in greater details because reasoning w.r.t. a knowledge base is computationally intensive. Nevertheless, a naive implementation of the $\mathcal{K}$-free rewriting rules is inefficient, and an approximation of the rewriting conditions is provided.

Since a variety of plans exist for a single user query, CARE implements a straightforward cost-based selection algorithm to choose a more efficient plan than the default plan. In addition, implementation details for algebraic operators in CARE are provided. An empirical study of CARE on query planning shows that the query optimization proposed in this work, together with a simple cost-based query planning strategy that derives from existing relational databases, can significantly improve query performance.

Chapter 6

Conclusions and Future Research

Object queries are an important class of queries that users pose over knowledge bases. This dissertation presents a framework for answering object queries, generalized as *assertion retrieval* over description logics knowledge bases, in which users are able to obtain specific details about objects that satisfy the given conditions. The key feature of this query paradigm is to compute projections of general concepts to make properties of objects syntactically explicit. To efficiently answer assertion queries, this work provides a basis for introducing efficient relational-style query processing. Particularly, details for caching concept assertions from previously computed answers are given, which can account for circumstances in which general knowledge base reasoning can be supplanted with reasoning without domain knowledge.

The task of assertion retrieval is inherently computationally intensive, because it entails a large volume of instance checking problems over the underlying knowledge bases. Although the relational-style query processing enables, in some circumstances, query answering without resorting to general reasoning, the standard approach to use instance checking are indispensable in general. To make instance checking more efficient, a novel algorithm called *ABox absorption* has been devised. The algorithm augments any $\mathcal{SHIQ}(\mathbb{D})$ knowledge base $\mathcal{K}$ with additional axioms that capture the interaction between individuals in $\mathcal{K}$. Such axioms are appropriately absorbed in a way such that they can be applied in a lazy fashion during tableau expansions. The ABox absorption algorithm then ensures that, for a given instance checking in the form of $a : C$, only those individuals that are necessary for deciding satisfiability are involved during tableau reasoning. Because the absorption relies on auxiliary concept names introduced to $\mathcal{K}$, dubbed *guards*, the effect of ABox absorption is also called *guarded reasoning.*

To validate the efficacy of the assertion retrieval framework, an empirical evaluation has been performed based on a prototypical system for answering assertion queries, i.e., CARE, which implements the proposed query optimization framework and ABox absorption. The experimental results on a number of diverse knowledge bases, including both synthesized and realistic ones, have corroborated the significance of the ideas presented in this research work.

6.1 Future Research

There are a number of possible extensions to the current framework for assertion retrieval. This section provides some thoughts on several interesting topics that can make this framework more practical and useful.

Refining Projection Descriptions A projection description defined as Pd currently serves two roles: a syntactic formatting specifier that controls the presentation of query answers, and a semantic parameter that captures some knowledge about an object deriving from the background knowledge base. It is possible to separate the two roles of a Pd such that additional artifacts can be introduced in Pd to deal with the syntactic aspects. The most important consequence is on the performance of extracting values from query answers. Consider the projection description $Pd = f? \sqcap g?$ used in some query. It is easy to see that query answers are in the form of $a : f = k_1 \sqcap g = k_2$. In the current framework, the extraction of a subexpression in the concept description of a necessitates reasoning, for instance, a cache scan that retrieves results that satisfy $f < k_3$ from these query answers will initiate a logical consequence check $a : (f = k_1 \sqcap g = k_2) \models a : f < k_3$. When Pd has syntactic extractors, a more straightforward and efficient way is to fetch the value of *the first field*, i.e., the f-values, in the query answers and perform comparisons properly, similar to what a relational database does.

Definability and Query Rewriting Section 5.2 presents the following condition for rewriting selection operations of the form $\sigma_C^{\mathcal{K}}(\mathtt{Q})$ into $\sigma_C^{\emptyset}(\mathtt{Q})$: $\{C\} \hookrightarrow \mathcal{L}_{\mathtt{Q}}$. If the condition is not met by C, then such a rewriting is not possible. However, observe that if the selection condition C can be superseded by another *equivalent* concept D, and it is the case that $\{D\} \hookrightarrow \mathcal{L}_{\mathtt{Q}}$, then a $\mathcal{K}$-free rewriting can be obtained. Indeed, the rewriting will be as follows: $\sigma_C^{\mathcal{K}}(Q) \equiv \sigma_D^{\emptyset}(Q)$. The problem of finding an appropriate replacement for C is characterized by the *Beth definability* theorem, and some recent work, e.g., [Ten Cate et al., 2011], has studied this problem in description logics.

Approximation Another direction for research is to consider approximation of TBoxes in a knowledge base with underlying expressive DL dialect. As an example, Zhou et al. [2012] presents a technique to obtain an upper and lower bound of a TBox in $\mathcal{K}$ such that the resulting approximated TBoxes are expressed by a less expressive language. Since reasoning with these approximated TBoxes has a lower complexity, answering the original query over these TBoxes is far more efficient. In the context of assertion retrieval, two new algebraic operators, *union* and *complement*, can be used to gather the answers that are beyond the lower bounds and that are in the upper bounds. Only these answers need to be reasoned w.r.t. $\mathcal{K}$, because the answers computed in the lower bound are already part of the final answers. In summary, a more efficient technology, including relational databases for the *DL-Lite* dialects, can be used to compute a superset of final answers, while the technology presented in this work can be used subsequently to find actual answers among a fairly small subset of potential answers, as opposed to dealing with all individuals in the ABox.

In addition to the aforementioned research directions, another practical and challenging direction is to work on identification of objects. Recall that the unique name assumption (UNA) stipulates that an object, i.e., an individual, is identified by its name. In many cases, this assumption does not hold; in particular, when the ABox data references multiple data sources during query evaluation. In relational databases, this is overcome by the use of *keys*. Nevertheless, a use of general keys in description logics makes the language undecidable [Toman and Weddell, 2005; Lutz et al., 2005]. Alternative ways to identifying objects in a knowledge base can include finding referring expressions for objects [Areces et al., 2008], i.e., computing a concept description to uniquely identify an object in a given context.

There are also a number of ways to improve the performance of the CARE system, particularly, many known optimizations for DL reasoning can be added to the implementation. It should be pointed out the architecture of CARE may be inappropriate for some optimizations. For instance, the binary retrieval technique may not be used when query evaluation is pipelined because the former saves execution time by combining several instance checking tasks, while the latter requires one instance checking request at a time.

References

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN 0-201-53771-0.

Renzo Angles and Claudio Gutierrez. The expressive power of SPARQL. In *Proceedings of the 7th International Conference on The Semantic Web*, ISWC '08, pages 114–129, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-88563-4. doi: http://dx.doi.org/10.1007/978-3-540-88564-1_8.

Carlos Areces, Alexander Koller, and Kristina Striegnitz. Referring expressions as formulas of description logic. In *Proceedings of the Fifth International Natural Language Generation Conference*, INLG '08, pages 42–49, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=1708322.1708332`.

Alessandro Artale, Diego Calvanese, Roman Kontchakov, and Michael Zakharyaschev. The DL-lite family and relations. *J. Artif. Int. Res.*, 36:1–69, September 2009. ISSN 1076-9757.

Franz Baader. Terminological cycles in a description logic with existential restrictions. In *Proceedings of the 18th international joint conference on Artificial intelligence*, IJCAI'03, pages 325–330, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc. URL `http://dl.acm.org/citation.cfm?id=1630659.1630707`.

Franz Baader, Bernhard Hollunder, Bernhard Nebel, Hans-Jrgen Profitlich, and Enrico Franconi. An empirical analysis of optimization techniques for terminological representation systems, or: Making KRIS get a move on. *Applied Artificial Intelligence*, 4: 109–132, 1994.

Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*, 2003. Cambridge University Press. ISBN 0-521-78176-0.

Franz Baader, Baris Sertkaya, and Anni-Yasmin Turhan. Computing the least common subsumer w.r.t. a background terminology. *J. App. Logic*, 5(3):392–420, 2007.

Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient RDF store over a relational database. In *ACM SIGMOD'13*, pages 121–132, New York, NY, USA, 2013. ACM.

Dan Brickley and R.V. Guha, editors. *RDF Vocabulary Description Language 1.0: RDF Schema*, February 2004. http://www.w3.org/TR/rdf-schema/.

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, and Riccardo Rosati. Ontologies and databases: The DL-Lite approach. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems*, volume 5689 of *Lecture Notes in Computer Science*, pages 255–356. Springer Berlin / Heidelberg, 2009.

Andrea Cal, Georg Gottlob, Thomas Lukasiewicz, Bruno Marnette, and Andreas Pieris. Datalog+/-: A family of logical knowledge representation and query languages for new applications. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pages 228–242. IEEE Computer Society, 2010. ISBN 978-0-7695-4114-3. doi: http://doi.ieeecomputersociety.org/10.1109/LICS.2010.27.

Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '98, pages 34–43, New York, NY, USA, 1998. ACM. ISBN 0-89791-996-3. doi: 10.1145/275487.275492. URL `http://doi.acm.org/10.1145/275487.275492`.

Julian Dolby, Achille Fokoue, Aditya Kalyanpur, Aaron Kershenbaum, Edith Schonberg, Kavitha Srinivas, and Li Ma. Scalable semantic retrieval through summarization and refinement. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pages 299–304, 2007.

Julian Dolby, Achille Fokoue, Aditya Kalyanpur, Li Ma, Edith Schonberg, Kavitha Srinivas, and Xingzhi Sun. Scalable grounded conjunctive query evaluation over large and expressive knowledge bases. In *Proceedings of the 7th International Conference on The Semantic Web*, ISWC '08, pages 403–418, Berlin, Heidelberg, 2008. Springer-Verlag.

ISBN 978-3-540-88563-4. doi: http://dx.doi.org/10.1007/978-3-540-88564-1_26. URL http://dx.doi.org/10.1007/978-3-540-88564-1_26.

Birte Glimm and Chimezie Ogbuji, editors. *SPARQL 1.1 Entailment Regime*, March 2013. http://www.w3.org/TR/sparql11-entailment/.

Jonathan D. Goldstein, Per-Ake Larson, and Jingren Zhou. Optimizing parameterized queries in a relational database management system, August 2006. US Patent, US8032522 B2.

Georg Gottlob and Thomas Schwentick. Rewriting ontological queries into small nonrecursive datalog programs. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012*, 2012.

Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Ontological queries: Rewriting and optimization. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 2–13, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-8959-6.

Goetz Graefe and David J. DeWitt. The EXODUS optimizer generator. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, SIGMOD '87, pages 160–172, New York, NY, USA, 1987. ACM. ISBN 0-89791-236-5. doi: 10.1145/38713.38734. URL http://doi.acm.org/10.1145/38713.38734.

Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 209–218. IEEE Computer Society, 1993. ISBN 0-8186-3570-3.

Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, June 1993. ISSN 1042-8143.

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semant.*, 3(2-3):158–182, October 2005. ISSN 1570-8268. doi: 10.1016/j.websem.2005.06.005. URL http://dx.doi.org/10.1016/j.websem.2005.06.005.

Volker Haarslev and Ralf Möller. On the scalability of description logic instance retrieval. *J. Autom. Reason.*, 41(2):99–142, August 2008. ISSN 0168-7433. doi: 10.1007/s10817-008-9104-7. URL http://dx.doi.org/10.1007/s10817-008-9104-7.

Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in starburst. *SIGMOD Rec.*, 18:377–388, June 1989. ISSN 0163-5808. doi: http://doi.acm.org/10.1145/66926.66962.

Alon Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4): 270–294, December 2001. ISSN 1066-8888. doi: 10.1007/s007780100054. URL `http://dx.doi.org/10.1007/s007780100054`.

Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011.

Stijn Heymans, Li Ma, Darko Anicic, Zhilei Ma, Nathalie Steinmetz, Yue Pan, Jing Mei, Achille Fokoue, Aditya Kalyanpur, Aaron Kershenbaum, Edith Schonberg, Kavitha Srinivas, Cristina Feier, Graham Hench, Branimir Wetzstein, and Uwe Keller. Ontology reasoning with large data repositories. In Martin Hepp, Pieter De Leenheer, Aldo de Moor, and York Sure, editors, *Ontology Management: Semantic Web, Semantic Web Services, and Business Applications*, volume 7 of *Semantic Web And Beyond Computing for Human Experience*, pages 89–128. Springer, 2008. URL `http://stijnheymans.net/pubs/reasoninglarge.pdf`.

Pascal Hitzler, Markus Krtzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph, editors. *OWL 2 Web Ontology Language Primer*, December 2012. http://www.w3.org/TR/owl2-primer/.

Ian Horrocks. *Optimising Tableaux Decision Procedures For Description Logics*. PhD thesis, the University of Manchester, 1997.

Ian Horrocks. Using an expressive description logic: FaCT or fiction? In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning*, pages 636–647, 1998.

Ian Horrocks and Ulrike Sattler. Optimised reasoning for shiq. In *ECAI'02*, pages 277–281, 2002.

Ian Horrocks and Stephan Tobies. Optimisation of terminological reasoning. In *Proceedings of the 2000 International Workshop on Description Logics*, pages 183–192, 2000a.

Ian Horrocks and Stephan Tobies. Reasoning with axioms: Theory and practice. In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning*, pages 285–296, 2000b.

Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive description logics. In *Proceedings of the 6th International Conference on Logic Programming and Automated Reasoning*, LPAR '99, pages 161–180, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66492-0. URL `http://dl.acm.org/citation.cfm?id=645709.664314`.

Alexander K. Hudek and Grant E. Weddell. Binary absorption in tableaux-based reasoning for description logics. In *Proceedings of the 2006 International Workshop on Description Logics*, 2006.

Ullrich Hustadt, Boris Motik, and Ulrike Sattler. Data complexity of reasoning in very expressive description logics. In *Proceedings of the 19th international joint conference on Artificial intelligence*, IJCAI'05, pages 466–471, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc. URL `http://dl.acm.org/citation.cfm?id=1642293.1642368`.

Vitaliy L. Khizder, David Toman, and Grant E. Weddell. Reasoning about duplicate elimination with description logic. In *Proceedings of the First International Conference on Computational Logic*, CL '00, pages 1017–1032, London, UK, 2000. Springer-Verlag. ISBN 3-540-67797-6.

Ilianna Kollia and Birte Glimm. Cost based query ordering over owl ontologies. In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, pages 231–246, 2012.

Vladimir Kolovski, Zhe Wu, and George Eadon. Optimizing enterprise-scale OWL 2 RL reasoning in a relational database system. In *International Semantic Web Conference (1)*, pages 436–452, 2010.

Roman Kontchakov, Carsten Lutz, David Toman, Frank Wolter, and Michael Zakharyaschev. The combined approach to query answering in DL-Lite. In Fangzhen Lin, Ulrike Sattler, and Miroslaw Truszczynski, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010*, 2010.

Roman Kontchakov, Carsten Lutz, David Toman, Frank Wolter, and Michael Zakharyaschev. The combined approach to ontology-based data access. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 2656–2661, 2011.

Carsten Lutz. Description logics with concrete domains—a survey. In *Advances in Modal Logics Volume 4*. King's College Publications, 2003.

Carsten Lutz. The complexity of conjunctive query answering in expressive description logics. In *Proceedings of the 4th international joint conference on Automated Reasoning*, IJCAR '08, pages 179–193, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-71069-1. doi: 10.1007/978-3-540-71070-7_16. URL `http://dx.doi.org/10.1007/978-3-540-71070-7_16`.

Carsten Lutz, Carlos Areces, Ian Horrocks, and Ulrike Sattler. Keys, nominals, and concrete domains. *J. Artif. Int. Res.*, 23:667–726, June 2005. ISSN 1076-9757. URL `http://dl.acm.org/citation.cfm?id=1622503.1622518`.

Alejandro Mallea, Marcelo Arenas, Aidan Hogan, and Axel Polleres. On blank nodes. In *Proceedings of the 10th international conference on The semantic web - Volume Part I*, ISWC'11, pages 421–437, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-25072-9. URL `http://dl.acm.org/citation.cfm?id=2063016.2063044`.

Frank Manola and Eric Miller, editors. *RDF Primer*, February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

Volker Markl. Query processing (in relational databases). In LING LIU and M.TAMER ÖZSU, editors, *Encyclopedia of Database Systems*, pages 2288–2293. Springer US, 2009. ISBN 978-0-387-35544-3. doi: 10.1007/978-0-387-39940-9_296. URL `http://dx.doi.org/10.1007/978-0-387-39940-9_296`.

Hector G. Molina, Jennifer Widom, and Jeffrey D. Ullman. *Database System Implementation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999. ISBN 0130402648.

Boris Motik, Rob Shearer, and Ian Horrocks. Hypertableau reasoning for description logics. *J. Artif. Intell. Res. (JAIR)*, 36:165–228, 2009.

Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz, editors. *OWL 2 Web Ontology Language Profiles*, December 2012. http://www.w3.org/TR/owl2-profiles/.

Thomas Neumann and Gerhard Weikum. The rdf-3x engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, February 2010. ISSN 1066-8888.

Esko Nuutila. Efficient transitive closure computation in large digraphs. *Acta Polytechnica Scandinavia: Math. Comput. Eng.*, 74:1–124, July 1995. ISSN 1237-2404.

Magdalena Ortiz, Diego Calvanese, and Thomas Eiter. Data complexity of query answering in expressive description logics via tableaux. *J. Autom. Reason.*, 41:61–98, July 2008. ISSN 0168-7433. doi: 10.1007/s10817-008-9102-9. URL `http://dl.acm.org/citation.cfm?id=1388522.1388532`.

Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009. ISSN 0362-5915. doi: 10.1145/1567274.1567278. URL `http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/1567274.1567278`.

Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. *Journal on data semantics*, X:133–173, 2008. URL `http://dl.acm.org/citation.cfm?id=1793934.1793939`.

Jeffrey Pound, Lubomir Stanchev, David Toman, and Grant E. Weddell. On ordering and indexing metadata for the semantic web. In *Proceedings of the 2008 International Workshop on Description Logics (DL2008)*. CEUR-WS.org, 2008.

Jeffrey Pound, David Toman, Grant Weddell, and Jiewen Wu. Concept Projection in Algebras for Computing Certain Answer Descriptions. In *22nd International Workshop on Description Logics*. CEUR-WS vol. 477, 2009.

Jeffrey Pound, David Toman, Grant E. Weddell, and Jiewen Wu. Query algebra and query optimization for concept assertion retrieval. In *Proceedings of the 23rd International Workshop on Description Logics (DL 2010)*, volume 573, 2010.

Jeffrey Pound, David Toman, Grant E. Weddell, and Jiewen Wu. An assertion retrieval algebra for object queries over knowledge bases. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 1051–1056, 2011.

Eric Prud'hommeaux and Andy Seaborne, editors. *SPARQL Query Language for RDF*, January 2008. http://www.w3.org/TR/rdf-sparql-query/.

Xiaolei Qian. Query folding. In *Proceedings of the Twelfth International Conference on Data Engineering*, ICDE '96, pages 48–55, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7240-4.

P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM. ISBN 0-89791-001-X. doi: 10.1145/582095.582099. URL `http://doi.acm.org/10.1145/582095.582099`.

Agricultural Research Service. U.S. Department of Agriculture national nutrient database for standard reference, release 25. Online, September 2012. http://www.ars.usda.gov/ba/bhnrc/ndl.

Evren Sirin, Bernardo Cuenca Grau, and Bijan Parsia. From wine to water: Optimizing description logic reasoning for nominals. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning*, pages 90–99, 2006.

Balder Ten Cate, Enrico Franconi, and İnanç Seylan. Beth definability in expressive description logics. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 1099–1106. AAAI Press, 2011. ISBN 978-1-57735-514-4. doi: 10.5591/978-1-57735-516-8/IJCAI11-188. URL `http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-188`.

David Toman and Grant Weddell. On reasoning about structural equality in xml: a description logic approach. *Theor. Comput. Sci.*, 336:181–203, May 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2004.10.036.

David Toman and Grant E. Weddell. *Fundamentals of Physical Design and Query Compilation.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

Dmitry Tsarkov and Ian Horrocks. Efficient reasoning with range and domain constraints. In *Proceedings of the 2004 International Workshop on Description Logics (DL2004)*, 2004.

Sebastian Wandelt and Ralf Möller. Towards abox modularization of semi-expressive description logics. *Applied Ontology*, 7(2):133–167, 2012.

Timo Weithöner, Thorsten Liebig, Marko Luther, and Sebastian Böhm. What's Wrong with OWL Benchmarks? In *Proc. of the Second Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2006)*, pages 101–114, Athens, GA, USA, November 2006.

Jiewen Wu and Volker Haarslev. Planning of axiom absorption. In *Proceedings of the 21st International Workshop on Description Logics (DL2008)*, volume 353. CEUR-WS.org, 2008.

Jiewen Wu, Alexander K. Hudek, David Toman, and Grant E. Weddell. Absorption for aboxes. In *Proceedings of the 2012 International Workshop on Description Logics*, volume 846, 2012a.

Jiewen Wu, Alexander K. Hudek, David Toman, and Grant E. Weddell. Assertion absorption in object queries over knowledge bases. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference*. AAAI Press, 2012b.

Jiewen Wu, Taras Kinash, David Toman, and Grant E. Weddell. Absorption for aboxes with local universal restrictions. In *Proceedings of the 2013 International Workshop on Description Logics*, pages 489–500, 2013.

Jian Zhou, Li Ma, Qiaoling Liu, Lei Zhang, Yong Yu, and Yue Pan. Minerva: A scalable OWL ontology storage and inference system. In *ASWC*, pages 429–443, 2006.

Yujiao Zhou, Bernardo Cuenca Grau, and Ian Horrocks. Efficient upper bound computation of query answers in expressive description logics. In *Proceedings of the 2012 International Workshop on Description Logics, DL-2012*, 2012.

APPENDICES

Appendix A

Correctness Proofs of the Rewriting Rules

In the following proofs, it is assumed that a given knowledge base $\mathcal{K}$ is *consistent*, hence, $\mathcal{K} \not\models a : \bot$ for every individual a occurring in $\mathcal{K}$. Recall that $[\![\mathtt{Q}]\!]_{\mathcal{K}}^{\mathbb{SI}}$ denote the set of concept assertions obtained by evaluating the algebraic expression $\mathtt{Q}$ over some knowledge base $\mathcal{K}$ with a given set of cached query results $\mathbb{SI}$. When $\mathcal{K}$ ($\mathbb{SI}$, respectively) is absent or empty (i.e., $\emptyset$), then query evaluation of $\mathtt{Q}$ does not depend on $\mathcal{K}$ ($\mathbb{SI}$, respectively). Observe that the order of concept assertions in $[\![\mathtt{Q}]\!]_{\mathcal{K}}^{\mathbb{SI}}$ is irrelevant for the proofs.

Rules in Figure 5.2

Proof. In the following we prove the transformation laws presented in Figure 5.2.

- (5.1). For any $a : (C_1 \sqcap C_2) \in [\![\mathtt{Q}_1 \bowtie \mathtt{Q}_2]\!]$, it holds that there are $a : C_1 \in [\![\mathtt{Q}_1]\!]$ and $a : C_2 \in [\![\mathtt{Q}_2]\!]$, which implies that $a : (C_1 \sqcap C_2) \in [\![\mathtt{Q}_2 \bowtie \mathtt{Q}_1]\!]$. The proof of the other direction is similar.

- (5.2). For any $a : (C_1 \sqcap C_2 \sqcap C_3) \in [\![\mathtt{Q}_1 \bowtie (\mathtt{Q}_2 \bowtie \mathtt{Q}_3)]\!]$, it holds that there are $a : C_i \in [\![\mathtt{Q}_i]\!]$, $i \in \{1, 2, 3\}$; therefore, it holds that $a : (C_1 \sqcap C_2) \in [\![\mathtt{Q}_1 \bowtie \mathtt{Q}_2]\!]$, which further implies that $a : (C_1 \sqcap C_2 \sqcap C_3) \in [\![(\mathtt{Q}_1 \bowtie \mathtt{Q}_2) \bowtie \mathtt{Q}_3]\!]$. The proof of the other direction is similar.

- (5.3). For any $a : D \in [\![S_1(\sigma_C^{\mathcal{K}}(\mathtt{Q}))]\!]_{\mathcal{K}}^{\mathbb{SI}}$, we have $a : D \in S_1$, such that there is $a : E \in [\![\mathtt{Q}]\!]_{\mathcal{K}}^{\mathbb{SI}}$, $\mathcal{K} \cup \{a : E\} \models a : C$ and $a : D \models a : E$. It also holds vacuously that

$\mathcal{K} \cup \{a : D\} \models a : E$ for $a : D \in S_1$, which implies $a : D \in [\![S_1(\mathtt{Q})]\!]^{\mathbb{SI}}_{\mathcal{K}}$; therefore, together with $\mathcal{K} \cup \{a : D\} \models a : C$, we can derive that $a : D \in [\![\sigma^{\mathcal{K}}_{C}(S_1(\mathtt{Q}))]\!]^{\mathbb{SI}}_{\mathcal{K}}$.

- (5.4). For any $a : D \in [\![\sigma^{\mathcal{K}}_{C_1}(\sigma^{\mathcal{K}}_{C_2}(\mathtt{Q}))]\!]^{\emptyset}_{\mathcal{K}}$, it holds that $a : D \in [\![\mathtt{Q}]\!]^{\emptyset}_{\mathcal{K}}$ such that $\mathcal{K} \cup \{a : D\} \models a : (C_1 \sqcap C_2)$; hence, it follows immediately that $a : D \in [\![\sigma^{\mathcal{K}}_{C_2}(\mathtt{Q})]\!]^{\emptyset}_{\mathcal{K}}$ and, further, $a : D \in [\![\sigma^{\mathcal{K}}_{C_2}(\sigma^{\mathcal{K}}_{C_1}(\mathtt{Q}))]\!]^{\emptyset}_{\mathcal{K}}$. The other direction is similar.

- (5.5). For any $a : D \in [\![\sigma^{\mathcal{K}}_{C_1}(\sigma^{\mathcal{K}}_{C_2}(\mathtt{Q}))]\!]^{\emptyset}_{\mathcal{K}}$, it holds that $a : D \in [\![\mathtt{Q}]\!]^{\emptyset}_{\mathcal{K}}$ such that $\mathcal{K} \cup \{a : D\} \models a : (C_1 \sqcap C_2)$; hence, it follows immediately that $a : D \in [\![\sigma^{\mathcal{K}}_{C_1 \sqcap C_2}(\mathtt{Q})]\!]^{\emptyset}_{\mathcal{K}}$. The other direction is similar.

- (5.6). This rule follows immediately from the semantics, given that C_1 and C_2 are semantically equivalent w.r.t. $\mathcal{K}$.

- (5.7). It suffices to prove that $a : \top \in [\![\pi^{\mathcal{K}}_{Pd_2}(\mathtt{Q})]\!]^{\emptyset}_{\mathcal{K}}$ iff $a : \top \in [\![\mathtt{Q}]\!]^{\emptyset}_{\mathcal{K}}$, where the concept $\top$ serves merely as a placeholder, i.e., these two subqueries compute exactly the same set of instances. The above claim, however, holds vacuously by the semantics.

- (5.8). For any $a : D \in [\![\pi^{\emptyset}_{Pd_1}(\pi^{\emptyset}_{Pd_2}(\mathtt{Q}))]\!]$, D is the most specific concept in $\mathcal{L}_{Pd_1}$ for some $a : E \in [\![\pi^{\emptyset}_{Pd_2}(\mathtt{Q})]\!]$ such that $\{a : E\} \models a : D$, where E is the most specific concept in $\mathcal{L}_{Pd_2}$ for some $a : F \in [\![\mathtt{Q}]\!]$ such that $\mathcal{K} \cup \{a : F\} \models a : E$. It is easy to see that $\{a : F\} \models a : D$. We then show that D is the most specific concept in $\mathcal{L}_{Pd_1}$ w.r.t. $a : F$. Assume, by way of contradiction, there is $D' \in \mathcal{L}_{Pd_1}$ such that $\{a : F\} \models a : D'$ and D' is more specific than D. It is evident that $E \in \mathcal{L}_{Pd_2}$ is more specific than D' because E is the most specific in $\mathcal{L}_{Pd_2}$ and $\mathcal{L}_{Pd_1} \subseteq \mathcal{L}_{Pd_2}$. Therefore, D', distinct from D, is the most specific concept in $\mathcal{L}_{Pd_1}$ for $a : E \in [\![\pi^{\emptyset}_{Pd_2}(\mathtt{Q})]\!]$ such that $\{a : E\} \models a : D'$: a contradiction. Consequently, $a : D \in [\![\pi^{\emptyset}_{Pd_1}(\mathtt{Q})]\!]$.

  The other direction. For $a : D \in [\![\pi^{\emptyset}_{Pd_1}(\mathtt{Q})]\!]$, it holds that D is the most specific concept in $\mathcal{L}_{Pd_1}$ for some $a : F \in [\![\mathtt{Q}]\!]$ such that $\{a : F\} \models a : D$. Now let $a : E \in [\![\pi^{\emptyset}_{Pd_2}(\mathtt{Q})]\!]$, where E is the most specific concept in $\mathcal{L}_{Pd_2}$ such that $\{a : F\} \models a : E$ (note that such E must exist given the existence of D). Because $\mathcal{L}_{Pd_1} \subseteq \mathcal{L}_{Pd_2}$ it follows that E is more specific than D, i.e., $\{a : E\} \models a : D$. We now show that D is the most specific concept in $\mathcal{L}_{Pd_1}$ w.r.t. $a : E$. Assume, by way of contradiction, that $D' \in \mathcal{L}_{Pd_1}$ is more specific than D and $\{a : E\} \models a : D'$. Then, it is easy to see that $\{a : F\} \models a : D'$, which implies that D', different from D, is also the most specific concept in $\mathcal{L}_{Pd_1}$ for some $a : F \in [\![\mathtt{Q}]\!]$: a contradiction. Hence, D is the most specific concept in $\mathcal{L}_{Pd_1}$, and, therefore, $a : D \in [\![\pi^{\emptyset}_{Pd_1}(\pi^{\emptyset}_{Pd_2}(\mathtt{Q}))]\!]$.

□

Lemma 5.1.1

Proof. Assume $\mathtt{Q}_1 = \pi^{\mathcal{K}}_{Pd}(\sigma^{\mathcal{K}}_{C}(P^{\mathcal{K}}))$ and $\mathtt{Q}_2 = \pi^{\mathcal{K}}_{Pd}(\sigma^{\mathcal{K}}_{C}((S_1(C_1) \bowtie \cdots \bowtie S_n(C_n))))$, we need to show that $[\![\mathtt{Q}_1]\!]^{\mathbb{SI}}_{\mathcal{K}} = [\![\mathtt{Q}_2]\!]^{\mathbb{SI}}_{\mathcal{K}}$, providing the following conditions hold: (i) $S_i = (D_i, Pd_i)$ and $S_i \in \mathbb{SI}$, (ii) $\mathcal{K} \models C \sqsubseteq (D_1 \sqcap ... \sqcap D_n)$, and (iii) $C_i = \lfloor\!\lfloor\{D \mid D \in \mathcal{L}_{Pd_i}, \mathcal{K} \models C \sqsubseteq D\}\rfloor\!\rfloor_{\mathcal{K}}$, for all $0 < i \leq n$.

We start by incrementally evaluating $\mathtt{Q}_2$. First, observe that

$$[\![S_1(C_1) \bowtie \cdots \bowtie S_n(C_n)]\!]^{\mathbb{SI}}_{\mathcal{K}} = \{a : E_1 \sqcap \cdots \sqcap E_n \mid a : E_i \in S_i, \\ \mathcal{K} \cup \{a : E_i\} \models a : C_i, a \text{ appears in } \mathcal{K}, 1 \leq i \leq n\}.$$

By evaluating the selection operator, we obtain

$$[\![\sigma^{\mathcal{K}}_{C}((S_1(C_1) \bowtie \cdots \bowtie S_n(C_n)))]\!]^{\mathbb{SI}}_{\mathcal{K}} = \{a : E_1 \sqcap \cdots \sqcap E_n \mid a : E_i \in S_i, \mathcal{K} \cup \{a : E_i\} \models a : C_i, \\ \mathcal{K} \cup \{a : E_1 \sqcap \cdots \sqcap E_n\} \models a : C, a \text{ appears in } \mathcal{K}, 1 \leq i \leq n\}.$$

Consequently, by evaluating the projection operator, we have

$$[\![\mathtt{Q}_2]\!]^{\mathbb{SI}}_{\mathcal{K}} = \{a : \lfloor\!\lfloor\{D \mid \mathcal{K} \cup \{a : \top\} \models a : D, D \in \mathcal{L}_{Pd}\}\rfloor\!\rfloor_{\mathcal{K}} \\ \mid a : E_i \in S_i, \mathcal{K} \cup \{a : E_i\} \models a : C_i, \mathcal{K} \cup \{a : E_1 \sqcap \cdots \sqcap E_n\} \models a : C, \\ a \text{ appears in } \mathcal{K}, 1 \leq i \leq n\} \tag{A.1}$$

Condition (i) states that $S_i = (D_i, Pd_i) :: Od_i$, therefore $a : E_i \in S_i$ implies that $a : E_i \in \{a : \{\lfloor\!\lfloor D \mid \mathcal{K} \cup \{a : \top\} \models a : D, D \in \mathcal{L}_{Pd}Pd\rfloor\!\rfloor_{\mathcal{K}}\} \mid \mathcal{K} \models a : D_i, \text{ a appears in } \mathcal{K}\}$, which ensures that $\mathcal{K} \cup \{a : \top\} \models a : E_i$. Also, $\mathcal{K} \cup \{a : \top\} \models a : E_1 \sqcap \cdots \sqcap E_n$. Note that the constraint $\mathcal{K} \models a : D_i$ in the definition of S_i can be overridden by the constraint $\mathcal{K} \models a : C$ because of Condition (ii). Therefore, these two conditions lead to a simplification of (A.1):

$$[\![\mathtt{Q}_2]\!]^{\mathbb{SI}}_{\mathcal{K}} = \{a : \lfloor\!\lfloor\{D \mid \mathcal{K} \cup \{a : \top\} \models a : D, D \in \mathcal{L}_{Pd}\}\rfloor\!\rfloor_{\mathcal{K}} \mid \mathcal{K} \cup \{a : \top\} \models a : C_i, \\ \mathcal{K} \cup \{a : \top\} \models a : C, a \text{ appears in } \mathcal{K}, 1 \leq i \leq n\} \tag{A.2}$$

Condition (iii) states that C_i is the minimum concept in $\mathcal{L}_{Pd_i}$ such that $\mathcal{K} \models C \sqsubseteq C_i$. Hence, (A.2) can be further simplified as follows.

$$[\![\mathtt{Q}_2]\!]^{\mathbb{SI}}_{\mathcal{K}} = \{a : \lfloor\!\lfloor\{D \mid \mathcal{K} \cup \{a : \top\} \models a : D, D \in \mathcal{L}_{Pd}\}\rfloor\!\rfloor_{\mathcal{K}} \\ \mid \mathcal{K} \cup \{a : \top\} \models a : C, a \text{ appears in } \mathcal{K}, 1 \leq i \leq n\} \tag{A.3}$$

For any consistent $\mathcal{K}$, $\mathcal{K} \models a : \top$ if a appears in $\mathcal{K}$; therefore, (A.3) is the same as (3.2), i.e., $[\![\mathtt{Q}_1]\!]^{\mathbb{SI}}_{\mathcal{K}} = [\![\mathtt{Q}_2]\!]^{\mathbb{SI}}_{\mathcal{K}}$, providing that the three conditions are satisfied.

□

Lemma 5.1.2

Proof. Let $\mathtt{Q}' = S_1(C_1) \bowtie \cdots \bowtie S_n(C_n)$, $\mathtt{Q}_1 = \pi^{\mathcal{K}}_{Pd}(\sigma^{\mathcal{K}}_{C}(\mathtt{Q}'))$, $\mathtt{Q}_2 = \pi^{\mathcal{K}}_{Pd}(\mathtt{Q}')$, we show that $[\![\mathtt{Q}_1]\!]^{\mathbb{SI}}_{\mathcal{K}} = [\![\mathtt{Q}_2]\!]^{\mathbb{SI}}_{\mathcal{K}}$ given that $\mathcal{K} \models (C_1 \sqcap \cdots \sqcap C_n) \sqsubseteq C$.

By definition,

$[\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}} = \{a : E_1 \sqcap \cdots \sqcap E_n \mid a : E_i \in S_i, \mathcal{K} \cup \{a : E_i\} \models a : C_i, a \text{ appears in } \mathcal{K}, 1 \leq i \leq n\}$;

hence it follows that

$[\![\sigma^{\mathcal{K}}_{C}(Q')]\!]^{\mathbb{SI}}_{\mathcal{K}} = \{a : E \mid \mathcal{K} \cup \{a : E\} \models a : C, a : E \in [\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}}\}$.

For any $a : E \in [\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}}$, we have $\mathcal{K} \cup \{a : E\} \models a : (C_1 \sqcap \cdots \sqcap C_n)$. Given that $\mathcal{K} \models (C_1 \sqcap \cdots \sqcap C_n) \sqsubseteq C$, we have $\mathcal{K} \cup \{a : E\} \models a : C$, therefore, $a : E \in [\![\sigma^{\mathcal{K}}_{C}(Q')]\!]^{\mathbb{SI}}_{\mathcal{K}}$. On the other hand, for any $a : E \in [\![\sigma^{\mathcal{K}}_{C}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\mathcal{K}}$, it must be the case that $a : E \in [\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}}$.

Therefore, $[\![Q']\!]^{\mathbb{SI}}_{\mathcal{K}} = [\![\sigma^{\mathcal{K}}_{C}(Q')]\!]^{\mathbb{SI}}_{\mathcal{K}}$, i.e.,$[\![Q_1]\!]^{\mathbb{SI}}_{\mathcal{K}} = [\![Q_2]\!]^{\mathbb{SI}}_{\mathcal{K}}$, given the condition that $\mathcal{K} \models (C_1 \sqcap \cdots \sqcap C_n) \sqsubseteq C$.

□

Lemma 5.1.3

Proof. We show a proof of a general case here, namely, $S_j(C_j)$ is generalized to any query $\mathtt{Q}'$. Observe that the final projection $\pi^{\mathcal{K}}_{Pd'}(\cdot)$ computes the same description for any qualifying objects for the given two subqueries. This observation reduces the proof to showing the following two queries, $\mathtt{Q}_1 = S_i(C_i) \bowtie \mathtt{Q}'$ and $\mathtt{Q}_2 = S_i(C_i \bowtie \pi^{\mathcal{K}}_{Pd}(\mathtt{Q}'))$, compute exactly the same set of qualifying objects, abstracting the descriptions of these objects. Consequently, it suffices to prove $a : \top \in [\![\mathtt{Q}_1]\!]^{\mathbb{SI}}_{\mathcal{K}}$ iff $a : \top \in [\![\mathtt{Q}_2]\!]^{\mathbb{SI}}_{\mathcal{K}}$, given that $\mathcal{L}_{Pd} \subseteq \mathcal{L}_{Pd_i}$. Note that in the subsequent proofs, the concept $\top$ in $a : \top$ is only used as a placeholder for the concept assertion, it does not, however, represent the resulting description of this object.

The $\rightarrow$ direction. Suppose $a : \top \in [\![\mathtt{Q}_1]\!]^{\mathbb{SI}}_{\mathcal{K}}$, then $a : \top \in S_i$, $a : C_i$ (w.r.t. S_i) and $a : \top \in [\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}}$. Because $a : \top \in [\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}}$ and projection does not disqualify objects, it follows that $a : \top \in [\![\pi^{\mathcal{K}}_{Pd}(Q')]\!]$. It also follows that $a : \top \in [\![C_i \sqcap \pi^{\mathcal{K}}_{Pd}(\mathtt{Q}')]\!]$ because of the constant query C_i. Finally, the condition $\mathcal{L}_{Pd} \subseteq \mathcal{L}_{Pd_i}$ ensures that any qualifying objects computed by $\pi^{\mathcal{K}}_{Pd}(\mathtt{Q}')$ will also be present in S_i, which contains more general answers. This condition, together with the facts that $a : C_i$ holds w.r.t. S_i and that $a : \top \in S_i$, ensures that $a : \top \in [\![S_i(C_i \bowtie \pi^{\mathcal{K}}_{Pd}(\mathtt{Q}'))]\!]$, i.e., $a : \top \in [\![\mathtt{Q}_2]\!]^{\mathbb{SI}}_{\mathcal{K}}$.

The $\leftarrow$ direction. Suppose $a : \top \in [\![\mathtt{Q}_2]\!]^{\mathbb{SI}}_{\mathcal{K}}$, then by definition, $a : \top \in S_i$, $a : C_i$ (w.r.t. S_i) and $a : \top \in [\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}}$, which implies $a : \top \in [\![S_i(C_i) \bowtie \mathtt{Q}']\!]$, i.e., $a : \top \in [\![\mathtt{Q}_1]\!]^{\mathbb{SI}}_{\mathcal{K}}$.

Since $\mathtt{Q}_1$ and $\mathtt{Q}_2$ computes the same set of objects, $[\![\pi^{\mathcal{K}}_{Pd'}(\mathtt{Q}_1)]\!]^{\mathbb{SI}}_{\mathcal{K}} = [\![\pi^{\mathcal{K}}_{Pd'}(\mathtt{Q}_2)]\!]^{\mathbb{SI}}_{\mathcal{K}}$. □

Theorem 5.2.2

We first prove the following lemmas:

Lemma A.0.1. *For any $a : D \in [\![\mathtt{Q}]\!]^{\mathbb{SI}}_{\mathcal{K}}$, where $\mathtt{Q}$ is pure, it holds that*

1. *$D \in \mathcal{L}_{\mathtt{Q}}$, and*
2. *$\forall D' \in \mathcal{L}_{\mathtt{Q}}$, if $\mathcal{K} \cup \{a : E\} \models a : D'$, where E is an arbitrary concept provided that $\mathcal{K} \cup \{a : E\}$ is consistent, then $\models D \sqsubseteq D'$.*

Intuitively, Lemma A.0.1 states that the concept in a query answer is the most specific concept in the corresponding representative language.

Proof. The proof of this claim is by structural induction on *pure* algebraic operations $\mathtt{Q}$.

- Cases $\mathtt{Q} = P^{\mathcal{K}}$. For any $a : D \in [\![\mathtt{Q}]\!]^{\mathbb{SI}}_{\mathcal{K}}$, $D = \top$, and $\mathcal{L}_{P^{\mathcal{K}}} = \{\top\}$; hence, the claim holds vacuously.

- Case $\mathtt{Q} = S_i(\mathtt{Q}_1)$ and $\mathcal{L}_{\mathtt{Q}} = \mathcal{L}_{Pd_i}$. For any $a : D \in [\![S_i(\mathtt{Q}_1)]\!]^{\mathbb{SI}}_{\mathcal{K}}$, $a : D \in S_i$, which, by Definition 17, implies that $D = \lfloor\!\lfloor S \rfloor\!\rfloor_{\mathcal{K}}$, where $S = \{D \mid D \in \mathcal{L}_{Pd_i}, \mathcal{K} \models a : D\}$. For any $D' \in \mathcal{L}_{Pd_i}$, if $\mathcal{K} \cup \{a : E\} \models a : D'$, it then follows that $\mathcal{K} \cup \{a : E\} \models a : D \sqcap D'$, i.e., $D \sqcap D'$ is satisfiable (because $\mathcal{K} \cup \{a : E\}$ is consistent). Thus, by Lemma 3.2.1, there is $D_3 \in \mathcal{L}_{Pd_i}$ and $\models D_3 \equiv D \sqcap D'$. Clearly, $D_3 \in S$. Assume, by way of contradiction, that $\not\models D \sqsubseteq D'$, then D_3 is strictly more specific than D; however, this is contradictory to the fact that D is the most specific concept in S by the definition of $D = \lfloor\!\lfloor S \rfloor\!\rfloor_{\mathcal{K}}$. Therefore, $\models D \sqsubseteq D'$.

- Case $\mathtt{Q} = \sigma^{\mathcal{K}}_C(\mathtt{Q}_1)$ and $\mathcal{L}_{\mathtt{Q}} = \mathcal{L}_{\mathtt{Q}_1}$. For any $a : D \in [\![\sigma^{\mathcal{K}}_C(\mathtt{Q}_1)]\!]^{\mathbb{SI}}_{\mathcal{K}}$, $a : D \in [\![\mathtt{Q}_1]\!]^{\mathbb{SI}}_{\mathcal{K}}$. By the induction hypothesis it holds that $D \in \mathcal{L}_{\mathtt{Q}_1}$ and that $\forall D' \in \mathcal{L}_{\mathtt{Q}_1}$, if $\mathcal{K} \cup \{a : E\} \models a : D'$ then $\models D \sqsubseteq D'$. Because $\mathcal{L}_{\mathtt{Q}} = \mathcal{L}_{\mathtt{Q}_1}$ and $\mathtt{Q}_1$ must be pure (as otherwise $\mathtt{Q}$ is impure), the previous two claims hold for $\mathcal{L}_{\mathtt{Q}}$ as well.

- Case $\mathtt{Q} = \pi^{\mathcal{K}}_{Pd}(\mathtt{Q}_1)$ and $\mathcal{L}_{\mathtt{Q}} = \mathcal{L}_{Pd}$. For any $a : D \in [\![\pi^{\mathcal{K}}_{Pd}(\mathtt{Q}_1)]\!]^{\mathbb{SI}}_{\mathcal{K}}$, by the semantics of projection (see Table 3.1), $D \in \mathcal{L}_{Pd}$; moreover, there is $a : C \in [\![\mathtt{Q}_1]\!]^{\mathbb{SI}}_{\mathcal{K}}$ such that $S = \{D_s \mid \mathcal{K} \cup \{a : C\} \models a : D_s, D_s \in \mathcal{L}_{Pd}\}$ and $D = \lfloor\!\lfloor S \rfloor\!\rfloor_{\mathcal{K}}$. For any $D' \in \mathcal{L}_{Pd}$ and $\mathcal{K} \cup \{a : E\} \models a : D'$, it holds that $\{D, D'\} \subseteq \mathcal{L}_{Pd}$. In addition,

$\mathcal{K} \cup \{a : C\} \cup \{a : E\} \models a : D \sqcap D'$, which implies $D \sqcap D'$ must be satisfiable because $\mathcal{K} \cup \{a : C\} \cup \{a : E\}$ is consistent by Lemma 3.5.1 for pure $\mathtt{Q}_1$. By Lemma 3.2.1, there is $D_3 \in \mathcal{L}_{Pd}$ and $\models D_3 \equiv D \sqcap D'$. Clearly, $D_3 \in S$. Assume, by way of contradiction, that $\not\models D \sqsubseteq D'$, then D_3 is strictly more specific than D; however, this is contradictory to the fact that D is the most specific concept in S by the definition of $D = \llcorner S \lrcorner_{\mathcal{K}}$. Therefore, $\models D \sqsubseteq D'$.

- Case $\mathtt{Q} = \mathtt{Q}_1 \bowtie \mathtt{Q}_2$ and $\mathcal{L}_{\mathtt{Q}} = \{D_1 \sqcap D_2 \mid D_1 \in \mathcal{L}_{\mathtt{Q}_1}, D_2 \in \mathcal{L}_{\mathtt{Q}_2}\}$. For any $a : D \in [\![\mathtt{Q}_1 \bowtie \mathtt{Q}_2]\!]^{\mathbb{SI}}_{\mathcal{K}}$, $D \equiv D_1 \sqcap D_2$ such that $a : D_1 \in [\![\mathtt{Q}_1]\!]^{\mathbb{SI}}_{\mathcal{K}}$ and $a : D_2 \in [\![\mathtt{Q}_2]\!]^{\mathbb{SI}}_{\mathcal{K}}$. By induction hypotheses, $D_i \in \mathcal{L}_{\mathtt{Q}_i}, i = \{1, 2\}$, it then follows that $D \in \mathcal{L}_{\mathtt{Q}}$. Furthermore, for any $D' \in \mathcal{L}_{\mathtt{Q}}$, if $\mathcal{K} \models a : D'$ and D' is defined to be $D'_1 \sqcap D'_2$, then $\models D \sqsubseteq D'$ also holds by the induction hypotheses that $\models D_i \sqsubseteq D'_i$, where D_i is an arbitrary concept in $\mathcal{L}_{\mathtt{Q}_i}$ and $\mathcal{K} \models a : D'_i, i = \{1, 2\}$.

Observe that the algebraic operations $\sigma^{\emptyset}_{C}(\mathtt{Q})$ and $\pi^{\emptyset}_{Pd}(\mathtt{Q})$ are just special cases when $\mathtt{Q} = \sigma^{\mathcal{K}}_{C}(\mathtt{Q}_1)$ and when $\mathtt{Q} = \pi^{\mathcal{K}}_{Pd}(\mathtt{Q}_1)$, respectively. $\square$

We can now demonstrate a proof of Theorem 5.2.2.

Proof. (1) The if direction.

Suppose $a : D \in [\![\sigma^{\mathcal{K}}_{C}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\mathcal{K}}$, then $a : D \in [\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}}$ and $\mathcal{K} \cup \{a : D\} \models a : C$. By the assumption $\{C\} \hookrightarrow \mathcal{L}_{\mathtt{Q}'}$, there is $C' \in \mathcal{L}_{\mathtt{Q}'}$ such that $\models C \equiv C'$. Because $\mathcal{K} \cup \{a : D\} \models a : C'$ and $\mathcal{K} \cup \{a : D\}$ is consistent ($a : D$ is a result of a *pure* query $\mathtt{Q}'$ and by Lemma 3.5.1), by Lemma A.0.1, $\models D \sqsubseteq C'$, or equivalently, $\models D \sqsubseteq C$, which implies $a : D \in [\![\sigma^{\emptyset}_{C}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\emptyset}$.

The only if direction.

Given $[\![\sigma^{\mathcal{K}}_{C}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\mathcal{K}} = [\![\sigma^{\emptyset}_{C}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\emptyset}$, for any $a : D \in [\![\sigma^{\emptyset}_{C}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\mathcal{K}}$, we have $a : D \in [\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}}$ and $\models D \sqsubseteq C$. In particular, $a : \top \in [\![\sigma^{\mathcal{K}}_{C}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\mathcal{K}}$, hence in this case $\models \top \sqsubseteq C$, or simply, $\models \top \equiv C$. It remains to show that $\top \in \mathcal{L}_{\mathtt{Q}'}$ for any pure $\mathtt{Q}'$. This can be demonstrated by the fact that if $a : \top \in [\![\sigma^{\mathcal{K}}_{C}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\mathcal{K}}$ then $a : \top \in [\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}}$ and Lemma A.0.1. Therefore, $\{C\} \hookrightarrow \mathcal{L}_{\mathtt{Q}'}$.

(2) The if direction.

For any $a : D \in [\![\pi^{\mathcal{K}}_{Pd}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\mathcal{K}}$, we have $D = \llcorner S \lrcorner_{\mathcal{K}}$ and S is defined to be $\{D \mid \mathcal{K} \cup \{a : C\} \models a : D, D \in \mathcal{L}_{Pd}\}$ for some $a : C \in [\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}}$. By the assumption $\mathcal{L}_{\pi^{\mathcal{K}}_{Pd}(\mathtt{Q}')} \hookrightarrow \mathcal{L}_{\mathtt{Q}'}$, i.e., $\mathcal{L}_{Pd} \hookrightarrow \mathcal{L}_{\mathtt{Q}'}$, we conclude that $\models D \equiv D'$ for some $D' \in \mathcal{L}_{\mathtt{Q}'}$. Because $\mathcal{K} \cup \{a : C\} \models a : D$ and $a : C$ is a result of the pure query $\mathtt{Q}'$, we have the consistency of $\mathcal{K} \cup \{a : C\}$ by Lemma 3.5.1. Consequently, by Lemma A.0.1, $C \in \mathcal{L}_{\mathtt{Q}'}$ and $\models C \sqsubseteq D'$ (clearly $\mathcal{K} \cup \{a :$

$C\} \models a : D')$. Therefore, S can be equivalently defined to be $\{D \mid \{a : C\} \models a : D, D \in \mathcal{L}_{Pd}\}$. Now we look at the reductions, i.e., $D = \lfloor\lfloor S \rfloor\rfloor_{\mathcal{K}}$, in which $\mathcal{K}$ is only exploited in the first reduction (see Definition 16), i.e., $\lfloor S \rfloor_{\mathcal{K}}$.

We now show how to completely remove $\mathcal{K}$ for the first reduction $\lfloor S \rfloor_{\mathcal{K}}$. Observe that $S \subseteq \mathcal{L}_{Pd}$ and, by $\mathcal{L}_{Pd} \hookrightarrow \mathcal{L}_{\mathtt{Q}'}$, we can replace S by $S_{\mathcal{L}} \subseteq \mathcal{L}_{\mathtt{Q}'}$ such that $S = S_{\mathcal{L}}$ up to equivalence, so, $\lfloor S \rfloor_{\mathcal{K}} = \lfloor S_{\mathcal{L}} \rfloor_{\mathcal{K}}$. We write $D_{\mathcal{L}} \in S_{\mathcal{L}}$ to denote the equivalent concept of $D \in S$. Given $D_{\mathcal{L}} \in S_{\mathcal{L}}$, for all $D'_{\mathcal{L}} \in S_{\mathcal{L}}$, we have $D_{\mathcal{L}} \in \mathcal{L}_{\mathtt{Q}'}$ and $D'_{\mathcal{L}} \in \mathcal{L}_{\mathtt{Q}'}$; additionally, $\mathtt{Q}'$ is pure. Obviously, $a : D_{\mathcal{L}} \in [\![\pi^{\mathcal{K}}_{Pd}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\mathcal{K}}$ by equivalence. Furthermore, by the definition of S (or equivalently $S_{\mathcal{L}}$), it is easy to see $\mathcal{K} \cup \{a : C\} \models a : D'_{\mathcal{L}}$ for some $a : C \in [\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}}$. By Lemma 3.5.1 $\mathcal{K} \cup \{a : C\}$ is consistent. With the above conditions, it follows from Lemma A.0.1 that $\models D_{\mathcal{L}} \sqsubseteq D'_{\mathcal{L}}$. Thus, by Definition 16, we can conclude that $\lfloor S_{\mathcal{L}} \rfloor_{\mathcal{K}}$ is equivalent to $\lfloor S_{\mathcal{L}} \rfloor$. Hence $\lfloor S \rfloor_{\mathcal{K}} = \lfloor S \rfloor_{\emptyset}$, i.e., $\mathcal{K}$ is removed in the reduction. Together with the second reduction, we have $D = \lfloor\lfloor S \rfloor\rfloor_{\mathcal{K}}$ iff $D = \lfloor\lfloor S \rfloor\rfloor_{\emptyset}$. The above proof guarantees that $\mathcal{K}$ is not needed for any $a : D \in [\![\pi^{\mathcal{K}}_{Pd}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\mathcal{K}}$, which means $a : D \in [\![\pi^{\emptyset}_{Pd}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\emptyset}$.

The only if direction.

We show that, for any Pd and pure $\mathtt{Q}'$, if $\mathcal{L}_{Pd} \not\hookrightarrow \mathcal{L}_{\mathtt{Q}'}$, then $[\![\pi^{\mathcal{K}}_{Pd}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\mathcal{K}} \neq [\![\pi^{\emptyset}_{Pd}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\emptyset}$. Observe that $a : \top$ is a possible result of any pure query $\mathtt{Q}'$, which also implies by Lemma A.0.1 that $\top$ is in $\mathcal{L}_{\mathtt{Q}'}$ for any pure $\mathtt{Q}'$. Assume $C \in \mathcal{L}_{Pd}$ and $C \notin \mathcal{L}_{\mathtt{Q}'}$ up to equivalence. Now let $\mathcal{K} \models \top \sqsubseteq C$ for some knowledge base $\mathcal{K}$. It is easy to observe the following: for some $a : \top \in [\![\mathtt{Q}']\!]^{\mathbb{SI}}_{\mathcal{K}}$, it holds that $a : C \in [\![\pi^{\mathcal{K}}_{Pd}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\mathcal{K}}$, but $a : C \notin [\![\pi^{\emptyset}_{Pd}(\mathtt{Q}')]\!]^{\mathbb{SI}}_{\emptyset}$.

□

Lemma 5.2.3

Proof. By Definition 30, the infinity of $\mathcal{L}_{\mathtt{Q}}$ is due to $\mathcal{L}_{Pd}$, which, by Definition 14, is on account of $\mathcal{L}^{\mathrm{TUP}}_{Pd}$ when $Pd = f?$, i.e., the infinity is ultimately owing to all the possible concrete domain values (i.e., strings in this work) for f. However, the definition of a finite approximate stipulates that the values of f in $\mathcal{L}^{\mathrm{fin}}_{\mathtt{Q}}$ is necessarily restricted to the admissible concrete domain values ($S_{\mathbb{D}}$) or $*$. Since the former has a finite number of strings by the definition of queries, $\mathcal{L}^{\mathrm{fin}}_{\mathtt{Q}}$ is finite as well.

To prove $\mathcal{L}_{\mathtt{Q}_1} \hookrightarrow \mathcal{L}_{\mathtt{Q}_2}$ iff $\mathcal{L}^{\mathrm{fin}}_{\mathtt{Q}_1} \hookrightarrow \mathcal{L}^{\mathrm{fin}}_{\mathtt{Q}_2}$, it suffices to consider only concepts that have subexpressions of the form $f = k'$, where $k' \notin S_{\mathbb{D}}$, because, intuitively, the restriction of $\mathcal{L}^{\mathrm{fin}}_{\mathtt{Q}}$ is to simply replace by $f = *$ any occurrences of $f = k'$ in $\mathcal{L}_{\mathtt{Q}}$, where $k' \notin S_{\mathbb{D}}$. Such concepts are called *variable concepts* for short in this proof.

It is easy to see the *only-if* direction holds. Assume for any variable concept $C_1 \in \mathcal{L}_{\mathtt{Q}_1}$ with a subconcept of the form $f = k'$ for some string $k' \notin S_{\mathbb{D}}$. Since there is $C_2 \in \mathcal{L}_{\mathtt{Q}_2}$ such that $\models C_1 \equiv C_2$, it must be the case that $f = k'$ is also a corresponding subconcept of C_2. Then, for the corresponding concept $C'_1 \in \mathcal{L}^{\text{fin}}_{\mathtt{Q}_1}$ of C_1, the subconcept of C_1 becomes $f = *$ and the rest remains the same by definition. It follows that there is $C'_2 \in \mathcal{L}^{\text{fin}}_{\mathtt{Q}_2}$ by simply replacing $f = k'$ in C_2 by $f = *$, and $\models C'_2 \equiv C'_1$.

The proof of *if* direction is also straightforward. As shown earlier in the proof, for $\mathcal{L}_{\mathtt{Q}}$ strings beyond $S_{\mathbb{D}}$ can *only* be introduced by $\mathcal{L}_{Pd}$ (i.e., when $\mathtt{Q} = S_i(\mathtt{Q}_1)$ or $\mathtt{Q} = \pi^{\mathcal{K}}_{Pd}(\mathtt{Q}_1)$) and $f?$ is part of Pd. For any variable concept $C_1 \in \mathcal{L}^{\text{fin}}_{\mathtt{Q}_1}$ with $f = *$ as a subconcept, there is $C_2 \in \mathcal{L}^{\text{fin}}_{\mathtt{Q}_2}$ with $f = *$ as a subconcept, by the condition that $\models C_1 \equiv C_2$. Consequently, there is at least one string $k' \notin S_{\mathbb{D}}$ ($k'' \notin S_{\mathbb{D}}$) and $f = k'$ ($f = k''$) is a subconcept of $C'_1 \in \mathcal{L}_{\mathtt{Q}_1}$ ($C'_2 \in \mathcal{L}_{\mathtt{Q}_2}$). By Definition 14, it must be the case that k' and k'' can be *any* possible strings. Therefore, by selecting $k'' \in \mathbb{D}$ such that $k'' = k'$, it holds that there is $C'_2 \in \mathcal{L}_{\mathtt{Q}_2}$ and $\models C'_1 \equiv C'_2$. □