Castor : a constraint-based SPARQL engine with active filter processing

Vianney Le Clement de Saint-Marcq

Université catholique de Louvain
Institute of ICT, Electronics
and Applied Mathematics
Louvain School of Engineering
Louvain-la-Neuve, Belgium

Université Claude Bernard Lyon I
Laboratoire d'Informatique, Image
et Systèmes d'information
École Doctorale InfoMaths
Lyon, France

Castor: a Constraint-based SPARQL Engine with Active Filter Processing

Vianney le Clément de Saint-Marcq

Thèse présentée en vue de l'obtention du grade de docteur en sciences de l'ingénieur
de l'Université catholique de Louvain, et de docteur (spécialité informatique)
de l'Université Claude Bernard Lyon I.

December 2013

Abstract

SPARQL is the standard query language for graphs of data in the Semantic Web. Evaluating queries is closely related to graph matching problems, and has been shown to be NP-hard. State-of-the-art SPARQL engines solve queries with traditional relational database technology. Such an approach works well for simple queries that provide a clearly defined starting point in the graph. However, queries encompassing the whole graph and involving complex filtering conditions do not scale well.

In this thesis we propose to solve SPARQL queries with Constraint Programming (CP). CP solves a combinatorial problem by exploiting the constraints of the problem to prune the search tree when looking for solutions. Such technique has been shown to work well for graph matching problems. We reformulate the SPARQL semantics by means of constraint satisfaction problems (CSPs). Based on this denotational semantics, we propose an operational semantics that can be used by off-the-shelf CP solvers.

Off-the-shelf CP solvers are not designed to handle the huge domains that come with Semantic Web databases though. To handle large databases, we introduce Castor, a new SPARQL engine embedding a specialized lightweight CP solver. Special care has been taken to avoid as much as possible data structures and algorithms whose time or space complexity are proportional to the database size.

Experimental evaluations on well-known benchmarks show the feasibility and efficiency of the approach. Castor is competitive with state-of-the-art SPARQL engines on simple queries, and outperforms them on complex queries where filters can be actively exploited during the search.

Résumé

SPARQL est le langage de requête standard pour les graphes de données du Web Sémantique. L'évaluation de requêtes est étroitement liée aux problèmes d'appariement de graphes. Il a été démontré que l'évaluation est NP-difficile. Les moteurs SPARQL de l'état-de-l'art résolvent les requêtes SPARQL en utilisant des techniques de bases de données traditionnelles. Cette approche est efficace pour les requêtes simples qui fournissent un point de départ précis dans le graphe. Par contre, les requêtes couvrant tout le graphe et impliquant des conditions de filtrage complexes ne passent pas bien à l'échelle.

Dans cette thèse, nous proposons de résoudre les requêtes SPARQL en utilisant la Programmation par Contraintes (CP). La CP résout un problème combinatoire en exploitant les contraintes du problème pour élaguer l'arbre de recherche quand elle cherche des solutions. Cette technique s'est montrée efficace pour les problèmes d'appariement de graphes. Nous reformulons la sémantique de SPARQL en termes de problèmes de satisfaction de contraintes (CSPs). Nous appuyant sur cette sémantique dénotationnelle, nous proposons une sémantique opérationnelle qui peut être utilisée pour résoudre des requêtes SPARQL avec des solveurs CP génériques.

Les solveurs CP génériques ne sont cependant pas conçus pour traiter les domaines immenses qui proviennent des base de données du Web Sémantique. Afin de mieux traiter ces masses de données, nous introduisons Castor, un nouveau moteur SPARQL incorporant un solveur CP léger et spécialisé. Nous avons apporté une attention particulière à éviter tant que possible les structures de données et algorithmes dont la complexité temporelle ou spatiale est proportionnelle à la taille de la base de données.

Des évaluations expérimentales sur des jeux d'essai connus ont montré la faisabilité et l'efficacité de l'approche. Castor est compétitif avec des moteurs SPARQL de l'état-de-l'art sur des requêtes simples, et les surpasse sur des requêtes complexes où les filtres peuvent être exploités activement pendant la recherche.

Acknowledgments

While there is only one author on the front page, the work presented in this thesis could not have been achieved without some substantial help. I am indebted to my advisors Yves Deville and Christine Solnon for their guidance during the last four years. Without their insight, patience and support, you, dear reader, would not be reading this thesis. I am also obliged to Pierre-Antoine Champin who introduced me to the Semantic Web. He is at the root of the idea of combining Constraint Programming and Semantic Web.

My gratitude goes to the jury members, Alain Mille, Amedeo Napoli, Charles Pecheur, Christophe Lecoutre, and Peter Van Roy for the careful reading of this document. Their comments contributed to put the final touch.

Very warm thanks go to my office mates Florence Massen and Jean-Baptiste Mairy. Their unswerving availability, whether for serious debates or funny chats, were drivers for my motivation. I am also grateful to Pierre Schaus for the detailed technical discussions. Thanks to Jean-Noël Monette, Julien Dupuis, Pham Quang Dung, Ho Trong Viet, Quoc Trung Bui, and Cyrille Dejemeppe for their inputs during our research meetings and the great moments we shared during conferences.

All INGI colleagues deserve a mention here. It was a great place to work and grow projects even outside the scope of the thesis. In particular, I would like to thank Antoine Cailliau, Sébastien Combéfis, and Simon Busard for all the work and joy we shared as part of the ACM Student Chapter or as researchers's representatives.

Members of the LIRIS lab, who always provided me a warm welcome, should not be forgotten. Special thanks go to Camille Combier, Stéphane Gosselin, and Loïc Blet.

Last, but certainly not least, I am deeply grateful to my parents, my brothers, and my friends. Without their constant loving support, I would never have reached the end of the journey.

This research was funded by the *Fonds National de la Recherche Scientifique* (FNRS).

Contents

Chapter 1

Introduction

The Internet has become the privileged means of looking for information in everyday's life. While the information abundantly available on the Web is increasingly accessible for human users, computers still have trouble making sense out of it. Developers have to rely on fuzzy machine learning techniques [CHM11] or site-specific APIs (e.g., Google APIs), or resort to writing a specialized parser that has to be updated on every site layout change.

The Semantic Web is an initiative of the World Wide Web Consortium (W3C) to enable sites to publish computer-readable data aside of the human-readable documents. Merging all published Semantic Web data results in one large global database. The global nature of the Semantic Web implies a much looser structure than traditional relational databases. A loose structure provides the needed flexibility to store unrelated data, but makes querying the database harder.

Amongst the various technologies related to the Semantic Web, we will focus on RDF and SPARQL. The Resource Description Framework (RDF) [KCM04] allows us to describe knowledge as a graph. Nodes are resources (e.g., people, objects, web pages, concepts, etc.) and literal values (e.g., numbers, strings, dates, etc.). Nodes are linked together with labeled edges to represent properties of a resource or relations between two resources. SPARQL [PS08] is the standard query language for RDF graphs. A SPARQL query basically consists of a pattern graph to be matched with the RDF graph containing the knowledge. The pattern graph may contain alternative or optional parts, as well as filtering conditions on the variables, making the evaluation more complex. Evaluating SPARQL queries in the general case has been shown to be PSPACE-complete [PAG09].

Example 1.1. Suppose we want to know which places are near each other, but have a big difference in average high temperature in August. The Wikipedia project contains a lot of facts about various places around the world. Those facts have been made available in the Semantic Web through the DBpedia project [Leh+13]. Figure 1.1 shows a subset of what can be found about Cotonou city. Now, we can write the

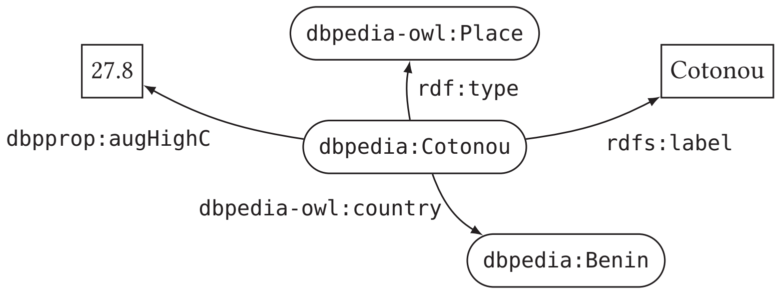


Figure 1.1: The DBpedia project [Leh+13] publishes facts found in Wikipedia as RDF data. This small excerpt shows information about the Cotonou city.

SPARQL query shown in listing 1.1 to solve our question. The query makes heavy use of filters and involves two initially undefined places. It is likely a complex query.

State-of-the-art SPARQL engines rely on relational database technology to solve queries. The RDF graph is usually stored in one big three-column table. Each row corresponds to an edge with its source node, destination node, and edge label. Every edge in the pattern graph of the query is mapped to a query on this table. The result sets then have to be joined together. Such operation can be costly if those intermediate result sets are large, e.g., if the pattern graph has no well-defined anchor in the RDF graph. Furthermore, filtering conditions involving different edges of the pattern graph can only be processed after the corresponding result sets have been joined. Hence, state-of-the-art engines do not perform well on complex queries. On the author's system, the query of example 1.1 was solved in 52 seconds using the state-of-the-art Virtuoso engine.[1]

Constraint Programming (CP) is a technique to solve hard combinatorial problems. Basically, it enumerates all solutions by traversing a search tree. To speed up such a search, it exploits the constraints of the problem to prune parts of the search tree that do not contain any solution. CP is an effective technique to solve graph matching problems [CDS09; ZDS10]. Filtering conditions can also be exploited early on during the search. Hence, we investigate whether CP could provide a good alternative to solve complex SPARQL queries. On the author's system, the same query of example 1.1 was solved in under 5 seconds with Castor, our CP-based engine.

Scope of the thesis

This thesis focuses on the database aspects of the Semantic Web, i.e., how data is stored and queried. We aim at exploring an alternative computation model for SPARQL

[1]On a Core i5 520M laptop with SSD and 8 GB RAM, running Arch Linux. The queried graph is a concatenation of the article-categories, category-labels, geo-coordinates, infobox-properties, infobox-property-definitions, instance-types, mappingbased-properties, persondata, skos-categories, and specific-mappingbased-properties datasets of the English DPpedia 3.8.

```
SELECT * WHERE {
  ?place1 rdf:type dbpedia-owl:Place ;
          dbpprop:augHighC ?temp1 ;
          geo:lat ?lat1 ;
          geo:long ?lon1 .
  ?place2 rdf:type dbpedia-owl:Place ;
          dbpprop:augHighC ?temp2 ;
          geo:lat ?lat2 ;
          geo:long ?lon2 .
  FILTER ( ?lat1 - ?lat2 < 0.1 && ?lat1 - ?lat2 > -0.1 &&
           ?lon1 - ?lon2 < 0.1 && ?lon1 - ?lon2 > -0.1 &&
           ?temp1 - ?temp2 > 5 &&
           ?place1 != ?place2 )
} LIMIT 10
```

Listing 1.1: SPARQL allows one to ask rich queries. Here, we are looking for places that are near each other, and whose average high temperature in August differ by more than 5°C.

queries, resulting in more efficient handling of complex queries.

Other Semantic Web aspects are left as future work. In particular, we do not consider reasoning on the data. With reasoning, one can infer more knowledge from the data, possibly changing the results of the queries. The conclusion chapter gives some leads on how the work of this thesis could be extended with reasoning.

Contributions

The first contribution of this thesis is the modeling of SPARQL queries in the CP framework. We reformulate the SPARQL semantics by means of Constraint Satisfaction Problems (CSPs), which is a declarative way to state combinatorial problems. The CSP reformulation extends the CSP framework slightly to accommodate for SPARQL features such as optional or alternative parts in the pattern graph. Based on the CSP reformulation, we propose an operational semantics that can be easily implemented in off-the-shelf solvers.

The second contribution is Castor, a SPARQL engine embedding a specialized lightweight CP solver. The CP models corresponding to SPARQL queries usually have few variables and constraints, but huge domains including all nodes of the RDF graph. Off-the-shelf solvers do not handle large domains well. Hence, we introduce a lightweight solver designed to cope with large domains. To achieve such a design goal, we avoid as much as possible data structures and algorithms whose time or space complexity is proportional to the domain sizes. All operations on the domains are performed in constant time. Constraints achieve forward-checking consistency, except where we can do better without maintaining costly internal structures.

Experimental evaluations on well-known benchmarks show the effectiveness of the approach and design. Castor is competitive with state-of-the-art engines on simple queries, thanks to its lightweight design. It is able to outperform them on complex queries involving filters on multiple variables, thanks to exploiting those filters during the search.

Outline

The first part gives background information about RDF and SPARQL (chapter 2), state-of-the-art SPARQL engines (chapter 3), and Constraint Programming (chapter 4). The second part details our contributions. Chapter 5 defines the reformulation of SPARQL queries by means of CSPs and gives a proof of correctness. It also shows the CP operational modeling. Chapter 6 explains the inner working of Castor and its lightweight solver. Finally, chapter 7 evaluates the effectiveness of our approach.

Publications

Parts of this thesis have been presented at the 17th International Conference on Principles and Practice of Constraint Programming [CDS11], at the 9th Extended Semantic Web Conference [Clé+12], and at the TRICS workshop collocated with the 19th International Conference on Principles and Practice of Constraint Programming [Clé+13].

[CDS11] Vianney le Clément de Saint-Marcq, Yves Deville, and Christine Solnon. "An Efficient Light Solver for Querying the Semantic Web". In: *Principles and Practice of Constraint Programming – CP 2011.* Ed. by Jimmy Lee. Vol. 6876. Lecture Notes in Computer Science. Springer, 2011, pp. 145–159. ISBN: 978-3-642-23785-0.

[Clé+12] Vianney le Clément de Saint-Marcq, Yves Deville, Christine Solnon, and Pierre-Antoine Champin. "Castor: A Constraint-Based SPARQL Engine with Active Filter Processing". In: *The Semantic Web: Research and Applications.* Ed. by Elena Simperl et al. Vol. 7295. Lecture Notes in Computer Science. Springer, 2012, pp. 391–405. ISBN: 978-3-642-30283-1.

[Clé+13] Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. "Sparse-Sets for Domain Implementation". In: *Techniques for implementing constraint programming systems (TRICS) workshop at CP 2013.* 2013.

The Castor system is available under the GPLv3 open-source license on the following web sites.

- `https://github.com/vianney/castor`
- `http://becool.info.ucl.ac.be/castor`

Part I

Background

Chapter 2

The Semantic Web

The main idea of the Semantic Web is best defined by Berners-Lee et al. [BHL01]:

> The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation.

The Semantic Web aims at complementing the web of documents by a web of data. As with the Hypertext Markup Language (HTML), data coming from various sources in the Semantic Web can be freely linked together.

To enable such *linked data*, publishers have to agree on common *vocabularies*. Many vocabularies have been proposed for various domains, e.g., for social networks [BM10], electronic publishing [Wei+98], bioinformatics [Bel+08], personal data management [Sce+07], geospatial data [BK11], etc.

Figure 2.1 shows the stack of technologies involved. The rest of this chapter focuses on the technologies used in the thesis.

2.1 Resource Description Framework (RDF)

The Resource Description Framework (RDF) [KCM04] allows one to model knowledge as a set of statements about things. Things described in RDF can be any arbitrary resource, ranging from real-world entities such as people, companies, objects, etc., to virtual things such as web pages, electronic documents, etc., or abstract concepts such as topics of interest, properties, categories, etc. In this section, we first provide a high-level overview of RDF (section 2.1.1). Then, we define RDF formally (section 2.1.2).

2.1.1 Overview

As a running example used throughout the thesis, fig. 2.2 shows the relations between some characters appearing in PhD Comics.[1] The graph encodes the following

[1] `http://www.phdcomics.com/`

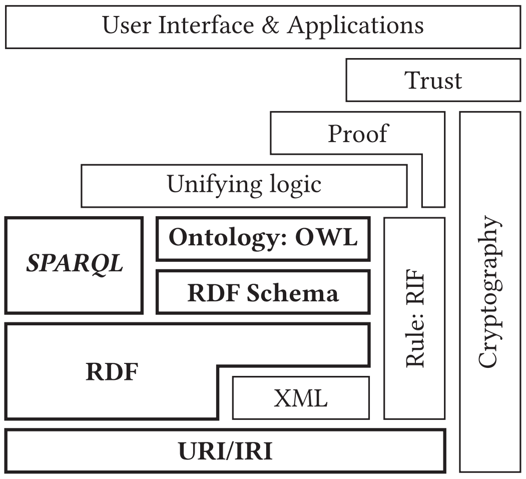


Figure 2.1: The Semantic Web includes a stack of technologies (schema from [Bra07]). This chapter covers the technologies written in bold. The thesis focuses on SPARQL and its underlying technologies.

knowledge by means of relations between resources:

> Tajel, Cecilia, and Mike are students who are respectively 29, 26, and 35 years old. Tajel and Cecilia know each other. Tajel also knows Mike, whose name is Michael Slackenerny. Mike knows Brian S. Smith, aged 56. Mike is interested in procrastination and free food. Smith is interested in research. Cecilia is interested in procrastination and maintains a blog on that topic at `http://www.phdcomics.com/blog.php`, created on 10 July 2005 at 8:20 AM. The blog's subjects are first about comics, and second about procrastination.

The "Friend of a Friend" (FOAF) vocabulary [BM10] is used to describe properties of the characters, such as name and age, group membership and topics of interest. The "Dublin Core Metadata Initiative" (DCMI) vocabulary [Wei+98] is used to describe properties on electronic documents, such as creation date and subject.

Resources are identified by Uniform Resource Identifiers (URIs) [BFM05]. A web address, such as `http://www.phdcomics.com/blog.php`, is an example of URI. URIs are defined as US-ASCII strings. As such encoding poses problems in an international environment, the RDF standard defines RDF URI references to be Unicode strings that can be mapped to ASCII URIs by a well-defined encoding. Such Unicode URIs were later standardized as Internationalized Resource Identifiers (IRIs) [DS05]. The SPARQL standard is defined using IRIs. This thesis considers all resources to be identified by IRIs. In the Semantic Web, IRIs are often abbreviated. For example, `foaf:name` is an abbreviation for `http://xmlns.com/foaf/0.1/name`, as defined in the header of listing 2.1.

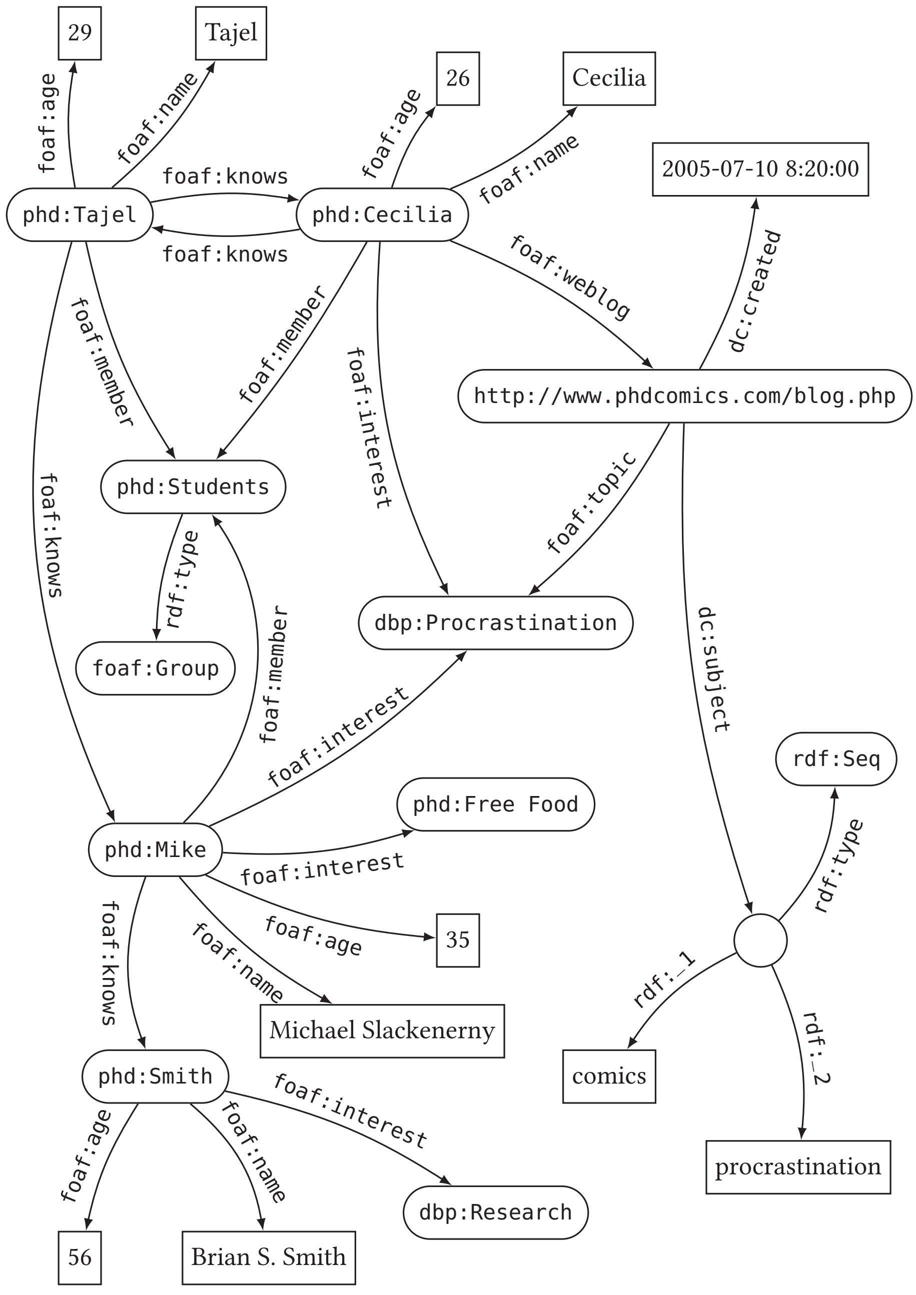


Figure 2.2: An RDF graph representing relations between PhD Comics characters. Rounded rectangles are IRIs, square rectangles are literals, circles are blank nodes. The datatypes of the literals are not shown in the graph.

An RDF graph may be encoded as a set of statements. A statement is encoded as a triple (subject, predicate, object). Such triple expresses a relation, defined by the predicate, between the subject and the object. For example, the triple (`phd:Tajel`, `foaf:member`, `phd:Students`) states that Tajel is a member of the students group. All three elements may be any IRI. A predicate may thus appear as a subject in another statement. Note that the graph and triples representations are equivalent. This means that isolated nodes in the graph are not allowed.

Besides IRIs, subjects and objects may also be blank nodes. A blank node is a resource whose name is not known. A blank node can also be considered as an existential quantifier, i.e., indicating something should be here, but we do not know what. Even though the RDF standard defines blank nodes as existential quantifiers, using blank nodes as anonymous resources is more widespread [ACM10]. In fig. 2.2, a blank node is used to group the subjects of the blog (comics and procrastination) in an ordered sequence.

Finally, objects may also be literals. A literal is a string with an optional type IRI or an optional language tag. The type IRI indicates how the string shall be interpreted. For example, ("29", `xsd:integer`) represents the integer number 29.

Listing 2.1 shows the triple representation of fig. 2.2 in N-Triples syntax. Full IRIs are enclosed in angled brackets, e.g., `<http://www.phdcomics.com/blog.php>`. Literals are surrounded by quotes, e.g., `"Tajel"`. The optional type IRI of the literal is appended with the ^^ operator, e.g., `"29"^^xsd:integer`. Other popular syntaxes include XML (the original syntax), Turtle (used in SPARQL queries), and RDFa (for embedding RDF data inside HTML documents).

An RDF graph may be assembled from various sources. IRIs serve as globally unique identifiers. The same IRI appearing in different RDF documents refers to the same node in the combined graph.

2.1.2 Formal Definition

Let $\mathbb{I}$, $\mathbb{B}$, $\mathbb{L}$, and $\mathbb{S}$ be pairwise disjoint infinite sets respectively representing IRIs, blank nodes, literals, and Unicode strings. The set of all *RDF terms* is denoted by $\mathbb{T} = \mathbb{I} \cup \mathbb{B} \cup \mathbb{L}$. These notations will be used throughout this document.

An RDF triple is a triple $(s, p, o) \in (\mathbb{I} \cup \mathbb{B}) \times \mathbb{I} \times (\mathbb{I} \cup \mathbb{B} \cup \mathbb{L})$. An RDF graph G is a set of RDF triples. We denote $\mathbb{T}_G$ the finite set of RDF terms appearing in graph G.

Definition 2.1. An *RDF graph* (or RDF dataset) is a finite set of triples $G \subset (\mathbb{I} \cup \mathbb{B}) \times \mathbb{I} \times (\mathbb{I} \cup \mathbb{B} \cup \mathbb{L})$.

Literals are partitioned in *plain literals* (denoted by $\mathbb{L}_p$) and *typed literals* (denoted by $\mathbb{L}_t$). Plain literals are strings with an optional language tag. Typed literals are strings with a mandatory type IRI. In contrast to plain literals, typed literals can be further interpreted according to the type IRI. For example, the typed literal `"29"^^xsd:integer` represents the integer number 29. Note that there are two kinds of strings: plain literals, e.g., `"Tajel"`, and typed strings, e.g., `"Cecilia"^^xsd:string`.

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dc:   <http://purl.org/dc/terms/> .
@prefix dbp:  <http://dbpedia.org/resource/> .
@prefix phd:  <http://phdcomics.com/#> .

phd:Students rdf:type foaf:Group .

# From Tajel’s foaf.rdf
phd:Tajel foaf:member phd:Students .
phd:Tajel foaf:name   "Tajel" .
phd:Tajel foaf:age    "29"^^xsd:integer .
phd:Tajel foaf:knows  phd:Cecilia .
phd:Tajel foaf:knows  phd:Mike .

# From Cecilia’s homepage
phd:Cecilia foaf:member   phd:Students .
phd:Cecilia foaf:name     "Cecilia"^^xsd:string .
phd:Cecilia foaf:age      "26"^^xsd:integer .
phd:Cecilia foaf:knows    phd:Tajel .
phd:Cecilia foaf:interest dbp:Procrastination .
phd:Cecilia foaf:weblog   <http://www.phdcomics.com/blog.php> .
<http://www.phdcomics.com/blog.php> foaf:topic dbp:Procrastination .

# From embedded RDFa on the page
<http://www.phdcomics.com/blog.php> dc:created
                                "2005-07-10T08:20:00"^^xsd:dateTime .
<http://www.phdcomics.com/blog.php> dc:subject _:a .
_:a rdf:type rdf:Seq .
_:a rdf:_1   "comics" .
_:a rdf:_2   "procrastination" .

# From Mike’s foaf.rdf
phd:Mike foaf:member   phd:Students .
phd:Mike foaf:name     "Michael Slackenerny" .
phd:Mike foaf:age      "35"^^xsd:decimal .
phd:Mike foaf:interest dbp:Procrastination .
phd:Mike foaf:interest phd:Free%20Food .
phd:Mike foaf:knows    phd:Smith .

# Generated from the department’s database
phd:Smith foaf:name     "Brian B. Smith" .
phd:Smith foaf:age      "56"^^xsd:integer .
phd:Smith foaf:interest dbp:Research .
```

Listing 2.1: Triples notation of the RDF graph depicted in fig. 2.2. The graph has been constructed by combining smaller graphs from various fictional sources. IRIs serve as globally unique identifiers.

A plain literal is a couple (s, l) where $s \in \mathbb{S}$ is the *lexical form* and $l \in \mathbb{S}$ the *language tag*, representing the language in which s is written. A *simple literal* is a plain literal with the empty string as language tag ($l = ""$). We denote $\mathbb{L}_{ps}$ the set of simple literals, and $\mathbb{L}_{pl}$ the set of plain literals that are not simple literals, i.e., plain literals with a non-empty language tag. A typed literal is a tuple (s, t) where $s \in \mathbb{S}$ is the lexical form and $t \in \mathbb{I}$ the *datatype*, defining how s should be interpreted.

To conveniently access the different parts of literals, we define the str, lang and datatype functions. Given a literal $a \in \mathbb{L}$, $\mathrm{str}(a)$ is the lexical form of a. For an IRI $i \in \mathbb{I}$, we also define $\mathrm{str}(i)$ to be the string representation of i. Given a plain literal $a = (s, l) \in \mathbb{L}_p$, $\mathrm{lang}(a) \triangleq l$ is the language tag of a. Given a typed literal $a = (s, t) \in \mathbb{L}_t$, $\mathrm{datatype}(a) \triangleq t$ is the datatype IRI of a.

RDF itself does not care about the interpretation of typed literals, but SPARQL handles some standard datatypes. We further partition the set of typed literals into strings ($\mathbb{L}_{ts}$), boolean values ($\mathbb{L}_{tb}$), numeric values ($\mathbb{L}_{tn}$), dates ($\mathbb{L}_{td}$) and other values ($\mathbb{L}_{to}$). The partitioning is based on the datatype IRI. The sets $\mathbb{L}_{tb}$, $\mathbb{L}_{tn}$ and $\mathbb{L}_{td}$ may contain ill-formed literals that cannot be interpreted. For example, $(\texttt{"z"}, \texttt{xsd:integer}) \in \mathbb{L}_{tn}$ is a valid literal, but not a valid number. Figure 2.3 shows a summary of the type hierarchy.

Let $a \in \mathbb{L}_t$, we denote $\mathrm{value}(a)$ the interpreted value of the typed literal a. If $a \in \mathbb{L}_{to}$ or if a is ill-formed, $\mathrm{value}(a)$ is the special value error. If $a \in \mathbb{L}_{ts}$, $\mathrm{value}(a) = \mathrm{str}(a)$. If $a \in \mathbb{L}_{tb} \cup \mathbb{L}_{tn} \cup \mathbb{L}_{td}$, $\mathrm{value}(a)$ is respectively the interpreted boolean, numeric or date value of $\mathrm{str}(a)$. Boolean values are denoted by true and false. Note that different lexical forms may have the same interpreted value. The value function is not injective.

The reverse operation, i.e., converting an interpreted value into an RDF term, is handled by the RDF function. For any typed literal a and interpreted value v, we have $\mathrm{RDF}(v) = a \Rightarrow \mathrm{value}(a) = v$. As a convenience for propagating errors, we define $\mathrm{RDF}(\mathrm{error}) = \mathrm{error}$.

2.2 Vocabularies and Inference

In its most basic form, a vocabulary consists of a set of RDF terms. Most vocabularies designed for reuse in the Semantic Web, come with additional information describing relations between vocabulary terms. For example, if a resource is a *member* of another resource, that other resource is a *group*. Such rules are often described with RDF Schema [Hay04] or the Web Ontology Language (OWL) [DS04].

RDF Schema allows to define a hierarchy of classes and to specify the domain and the range of properties. For example, fig. 2.4 shows a part of the RDF Schema rules for the FOAF vocabulary. The rules state the following:

- If x is the subject or object of a triple with predicate `foaf:knows`, then we can infer the triple $(x, \texttt{rdf:type}, \texttt{foaf:Person})$, i.e., resources knowing each other are persons.

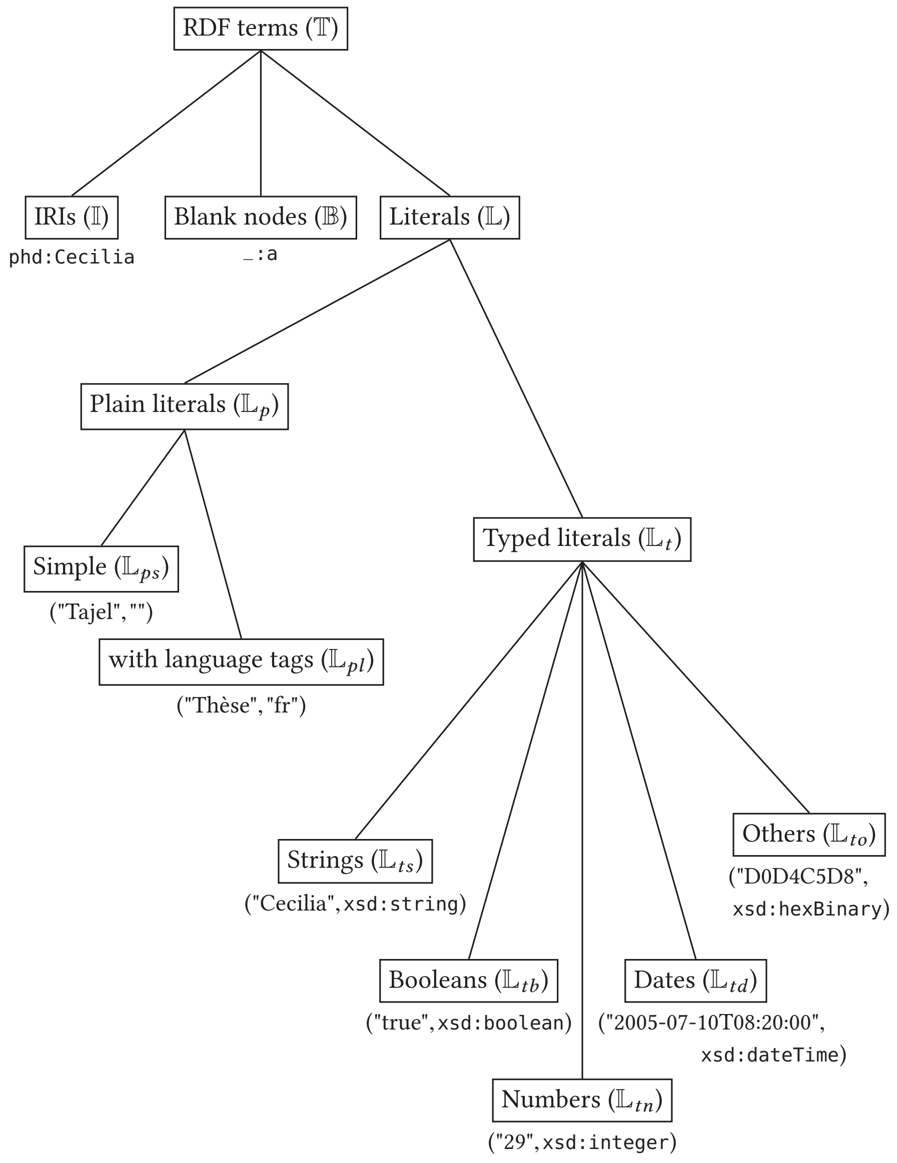

RDF terms ($\mathbb{T}$)
IRIs ($\mathbb{I}$)
phd:Cecilia
Blank nodes ($\mathbb{B}$)
_:a
Literals ($\mathbb{L}$)
Plain literals ($\mathbb{L}_p$)
Simple ($\mathbb{L}_{ps}$)
("Tajel", "")
with language tags ($\mathbb{L}_{pl}$)
("Thèse", "fr")
Typed literals ($\mathbb{L}_t$)
Strings ($\mathbb{L}_{ts}$)
("Cecilia", xsd:string)
Booleans ($\mathbb{L}_{tb}$)
("true", xsd:boolean)
Numbers ($\mathbb{L}_{tn}$)
("29", xsd:integer)
Dates ($\mathbb{L}_{td}$)
("2005-07-10T08:20:00", xsd:dateTime)
Others ($\mathbb{L}_{to}$)
("D0D4C5D8", xsd:hexBinary)

Figure 2.3: RDF terms are partitioned into a type hierarchy. Examples are shown below the classes.

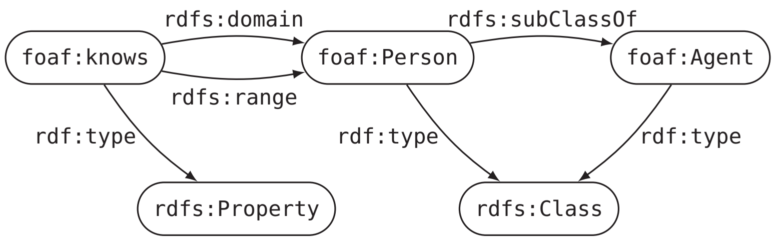


Figure 2.4: With RDF Schema, one can specify that if two resources *know* each other, then both are *persons*. Every *person* is also an *agent*. This graph is part of the FOAF specification [BM10].

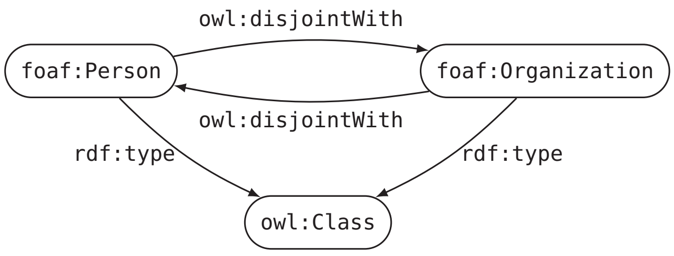


Figure 2.5: With OWL, one can specify that a resource cannot be a *person* and an *organization* at the same time. This graph is part of the FOAF specification [BM10].

- If we have (x, `rdf:type`, `foaf:Person`), then we can infer the triple (x, `rdf:type`, `foaf:Agent`), i.e., a person is an agent.

RDF Schema has if-semantics [Hor05], i.e., it cannot add contradictory knowledge. An RDF graph can never be invalid in such semantics.

OWL extends RDF Schema to allow rules such as equivalences between classes or properties, mutual exclusions, cardinality constraints, etc. OWL has iff-semantics, meaning some RDF graphs may be incoherent. For example, fig. 2.5 states that an RDF graph may not contain both (x, `rdf:type`, `foaf:Person`) and (x, `rdf:type`, `foaf:Organization`) at the same time for any resource x.

RDF defines the notion of *entailment* [Hay04]. A graph G_1 entails a graph G_2, noted by $G_1 \models G_2$, if G_1 contains more knowledge than G_2. Formally, entailment is defined by means of interpretations. The exact definition depends on the used entailment regime, e.g., RDF Schema or OWL rules. When no such regime is used, i.e., when *simple entailment* is used, $G_1 \models G_2$ if and only if there exists a homomorphism from G_2 into G_1. A homomorphism exists if there exists a mapping μ from the blank nodes of G_2 to the terms of G_1 such that $\mu[G_2] \subseteq G_1$, where $\mu[G_2]$ is the graph obtained by replacing the blank nodes of G_2 by their value in μ [Bag05]. From the entailment perspective, any blank node is an existential quantifier. If there are no blank nodes in G_2, $G_1 \models G_2 \Leftrightarrow G_2 \subseteq G_1$.

Example 2.1. Consider the graph G shown in fig. 2.2, and the graphs G_1 consisting of the single triple (`phd:Tajel`, `foaf:member`, `phd:Students`), and G_2 consisting of the single triple (b, `foaf:member`, `phd:Students`), where b is a blank node. We have $G \models G_1$, because the triple set of G_1 is a subset of the triple set of G. Let $\mu = \{ (b, \texttt{phd:Tajel}) \}$, we have $\mu[G_2] \subset G$, and thus $G \models G_2$. Note that other mappings μ are possible.

For the RDF Schema and OWL entailment regimes, $G_1 \models G_2$ if and only if G_2 is entailed by the deductive closure of G_1 with respect to the considered regime. The deductive closure of a graph is obtained by adding all triples inferred by the RDF Schema or OWL rules. In practice, the deductive closure is not necessarily computed entirely.

Example 2.2. Using the RDF Schema entailment regime with the rules of fig. 2.4, the graph consisting solely of the triple (`phd:Tajel`, `rdf:type`, `foaf:Agent`) is entailed by the graph of fig. 2.2. Because `phd:Tajel` is the subject of a triple with predicate `foaf:knows`, we infer that Tajel is of type `foaf:Person`. As every person is an agent, we further infer that Tajel is of type `foaf:Agent`.

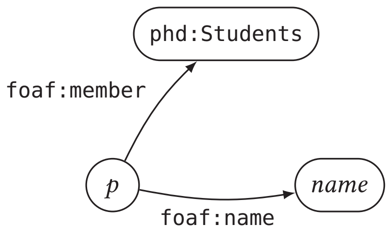
Verifying that $G_1 \models G_2$ under simple entailment or the RDF Schema regime is NP-complete, except when there are no blank nodes in G_2 [Hor05]. Entailment under the OWL regime is undecidable. OWL DL is a decidable subset of OWL designed to circumvent this drawback.

2.3 SPARQL Query Language

SPARQL [PS08] is a query language for RDF. In its simplest form, a query is a set of triple patterns, i.e., triples where elements may be replaced by variables. Figure 2.6 shows an example querying the names of all PhD students. The set of triple patterns, called the basic graph pattern, defines a pattern graph that has to be matched with the target dataset. A solution consists of a mapping of the variables of the pattern graph to terms of the dataset. This is similar to the entailment of RDF graphs explained in section 2.2. However, we are now interested in the mappings themselves instead of merely their existence.

```
SELECT ?name WHERE {
  ?p foaf:member phd:Students .
  ?p foaf:name   ?name .
}
```

(a) SPARQL query



(b) Associated pattern graph

Figure 2.6: Simple SPARQL query for the dataset shown in fig. 2.2. The query returns the names of all the PhD students.

SPARQL queries combine basic graph patterns into compound patterns with composition, optional or alternative parts. Filters add constraints on the variables. Similarly to SQL, from which SPARQL borrows its syntax, the results may be sorted, projected, filtered from duplicates, etc.

This section presents the abstract syntax (section 2.3.1) and the semantics (section 2.3.2) of the SPARQL language, based on Pérez et al. [PAG09]. In order to cover a broader part of the SPARQL specification, we define the full semantics of the expressions, and explain the solution modifiers. Compared to the official W3C recommendation [PS08], the following definition makes some simplifying assumptions without restricting the expressiveness of the language, as explained by Angles and Gutierrez [AG08]. For the sake of readability, this chapter uses set semantics instead of the bag semantics described in the recommendation. Such simplifications are also done by Pérez et al. [PAG09]. The results can be easily extended to bag semantics.

2.3.1 SPARQL Syntax

A SPARQL query consists of two parts: a graph pattern and solution modifiers. The graph pattern is to be matched with the RDF graph. The resulting solution set is transformed into a list according to the solution modifiers.

To avoid dealing with parsing specificities, we present here an algebraic syntax for SPARQL queries. We first present expressions that may appear in various parts of the query. Then we define graph patterns and solution modifiers.

Let $\mathbb{V}$ be an infinite set representing variables. The set of variables is disjoint from the set of RDF terms, i.e., $\mathbb{V} \cap \mathbb{T} = \emptyset$.

Definition 2.2. An *expression* is recursively defined as follows.

- If $a \in \mathbb{I} \cup \mathbb{L}$, then (a) is an expression.
- If $x \in \mathbb{V}$, then (x) and $(\mathsf{bound}(x))$ are expressions.
- If E is an expression, then $(\neg E)$, $(\mathsf{isIRI}(E))$, $(\mathsf{isBlank}(E))$, $(\mathsf{isLiteral}(E))$, $(\mathsf{str}(E))$, $(\mathsf{lang}(E))$ and $(\mathsf{datatype}(E))$ are expressions.
- If E_1 and E_2 are expressions, then $(E_1 \wedge E_2)$, $(E_1 \vee E_2)$, $(E_1 = E_2)$, $(E_1 \neq E_2)$, $(E_1 < E_2)$, $(E_1 \leqslant E_2)$, $(E_1 > E_2)$, $(E_1 \geqslant E_2)$, $(\mathsf{sameTerm}(E_1,E_2))$, $(E_1 * E_2)$, (E_1/E_2), $(E_1 + E_2)$ and $(E_1 - E_2)$ are expressions.

To summarize, an expression consists of IRIs, literals and variables, but not blank nodes, composed together with logical connectives, comparison operators, arithmetic operators, unary functions and unary and binary predicates. The SPARQL standard also defines the langMatches and regEx predicates, as well as type casting operators. For concision, we do not consider such operators in this chapter.

The building block of a graph pattern is a triple pattern. A triple pattern is an RDF triple, where components may be replaced by variables. A set of triple patterns is called a basic graph pattern (BGP). BGPs may be composed together with binary operators to build more complex graph patterns. Patterns may also be filtered by expressions.

Let P be a graph pattern, we denote $\text{vars}(P)$ the set of variables appearing in P. Similarly, if E is an expression, we denote $\text{vars}(E)$ the set of variables appearing in E.

Definition 2.3. A *triple pattern* is a tuple (s,p,o) where $s \in \mathbb{I} \cup \mathbb{V}$, $p \in \mathbb{I} \cup \mathbb{V}$, and $o \in \mathbb{I} \cup \mathbb{L} \cup \mathbb{V}$.

Definition 2.4. A *graph pattern* is recursively defined as follows.

1. $P \subset (\mathbb{I} \cup \mathbb{V}) \times (\mathbb{I} \cup \mathbb{V}) \times (\mathbb{I} \cup \mathbb{L} \cup \mathbb{V})$ is a *basic* graph pattern, i.e., a set of triple patterns.

2. Let P_1 and P_2 be graph patterns. $(P_1 \text{ AND } P_2)$, $(P_1 \text{ UNION } P_2)$, $(P_1 \text{ DIFF } P_2)$ and $(P_1 \text{ OPT } P_2)$ are *compound* graph patterns.

3. Let P be a graph pattern and E be an expression, such that $\text{vars}(E) \subseteq \text{vars}(P)$, $(P \text{ FILTER } E)$ is a *constrained* graph pattern.

Without loss of generality, we have excluded blank nodes from appearing in triple patterns. Blank nodes appearing in a query are considered as existential quantifiers and may be replaced by fresh variables [Mal+11].

The AND, UNION and OPT operators map respectively to the period (`.`), `UNION` and `OPTIONAL` keywords in SPARQL. The W3C recommendation does not define a DIFF operator. However, such operator can be obtained by combining OPT and FILTER operators as shown in Angles and Gutierrez [AG08].

For reasons of simplicity, we also restrict the scope of a filter expressions to the pattern it constrains. Such approach is also followed by Pérez et al. [PAG09] and does not alter the expressive power of the language [AG08].

Solution modifiers determine which variables of which solutions should be returned and in what order. A set of solution modifiers may contain at most one solution modifier of each type.

Definition 2.5. A *solution modifier* is one of

- $\text{PROJECT}(X)$, where $X \subset \mathbb{V}$,
- DISTINCT,
- $\text{LIMIT}(n)$, where $n \in \mathbb{N}$,
- $\text{OFFSET}(n)$, where $n \in \mathbb{N}$,
- $\text{ORDER}(\langle O \rangle)$, where $\langle O \rangle$ is a sequence of couples (E,D) with E an expression and $D \in \{ \text{ASC}, \text{DESC} \}$.

A graph pattern and a set of modifiers together is a complete SPARQL query. The W3C recommendation also specifies three query forms: SELECT, ASK and CONSTRUCT.

The SELECT form returns the list of solutions as mappings between variables and RDF terms. The ASK form implies the LIMIT(1) solution modifier. The result is “yes” if there is a solution and “no” otherwise. The CONSTRUCT form generates an RDF graph for every solution, by replacing variables with their values in a template graph. Query forms are mostly cosmetic and do not alter the way the query is solved. As such, we restrict ourselves to the SELECT query form.

Definition 2.6. A *query* is a couple $Q = (P, M)$, with P a graph pattern and M a set of solution modifiers. A *query instance* is a couple (Q, G), with Q a query and G an RDF graph.

2.3.2 SPARQL Semantics

A solution of a query instance is an assignment of variables to RDF values. The solutions of a query instance are defined in two steps. Evaluating the graph pattern, results in a set of solutions. The solution modifiers determine how to transform this set into a list.

Definition 2.7. A *solution mapping* is a partial function $\mu : \mathbb{V} \to \mathbb{T}$. The domain of the mapping is denoted by $\text{dom}(\mu)$.

A solution mapping is also represented as a set of assignments (x, v). Set operations, like the union of two sets, can be applied on solution mappings, provided the operands are compatible.

Definition 2.8. Two mappings μ_1 and μ_2 are said to be *compatible*, denoted by $\mu_1 \sim \mu_2$, if $\forall x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2), \mu_1(x) = \mu_2(x)$.

Two mappings are compatible if they agree on their shared variables. Intuitively, one mapping can be *extended* into the other by assigning more variables. Note that the $\sim$ relation is reflexive and symmetric, but not transitive.

Before defining the evaluation of a graph pattern, we show the evaluation of an expression. Expressions are evaluated when handling constrained patterns and solution modifiers. Given a solution mapping μ, we first substitute each variable assigned by μ by its assigned value. Then, we get the value of the resulting expression. Any variable left is unbound and results in an error.

Definition 2.9. The *substitution* of a solution mapping μ in an expression E, denoted by $\mu[E]$, is the expression obtained by applying the following operations on E for any $x \in \text{dom}(\mu)$.

1. Replace each occurrence of $\mathsf{bound}(x)$ by the value RDF(true).
2. Replace each occurrence of x that is not the operand of a $\mathsf{bound}()$ predicate by the value $\mu(x)$.

As the evaluation of an expression may result in an error, a three-state Boolean logic is used. The states true and false have their usual meaning. An additional state

$a \wedge b$	true	false	error
true	true	false	error
false	false	false	false
error	error	false	error

$a \vee b$	true	false	error
true	true	true	true
false	true	false	error
error	true	error	error

Table 2.1: Truth tables for logical connectives in a three-state boolean logic. Note that the error propagation may seem counter-intuitive.

error indicates an evaluation error. Negating error has no effect, i.e., $\neg$error = error. The semantics of the $\wedge$ and $\vee$ connectives are given by the truth tables in table 2.1.

The value of an expression E, denoted by $[\![E]\!]$, is either an RDF term or error. Any RDF term can be used as a Boolean predicate. The conversion of an RDF term v into a Boolean value is called the effective Boolean value of the RDF term, denoted by EBV(v). For convenience, we also define EBV(error) = error.

Definition 2.10. The *effective Boolean value* of a value $v \in \mathbb{T} \cup \{ \text{error} \}$, denoted by EBV($v$), is

$$\mathrm{EBV}(v) \triangleq \begin{cases} \text{true} & \text{if } (v \in \mathbb{L}_{tb} \wedge \text{value}(v) = \text{true}) \vee \\ & \quad (v \in \mathbb{L}_{tn} \wedge \text{value}(v) \notin \{ 0, \text{NaN} \}) \vee \\ & \quad (v \in \mathbb{L}_{p} \cup \mathbb{L}_{ts} \wedge \text{str}(v) \neq \text{""}) \\ \text{error} & \text{if } t \notin \mathbb{L}_{tb} \cup \mathbb{L}_{tn} \cup \mathbb{L}_{p} \cup \mathbb{L}_{ts} \\ \text{false} & \text{otherwise} \end{cases}$$

where NaN is the special not-a-number value of standard IEEE 754 floating point arithmetic.

Definition 2.11. The *value* of an expression E, denoted by $[\![E]\!]$, is an RDF term or an error, recursively defined as follows.

1. If $E \equiv (a)$, where $a \in \mathbb{T}$, $[\![E]\!] \triangleq a$.

2. If $E \equiv (x)$, where $x \in \mathbb{V}$, $[\![E]\!] \triangleq$ error.

3. If $E \equiv (\neg E')$, $[\![E]\!] \triangleq \mathrm{RDF}\left(\neg\, \mathrm{EBV}([\![E']\!])\right)$.

4. If $E \equiv (E_1 \bullet E_2)$, where $\bullet \in \{ \wedge, \vee \}$, $[\![E]\!] \triangleq \mathrm{RDF}\left(\mathrm{EBV}([\![E_1]\!]) \bullet \mathrm{EBV}([\![E_2]\!])\right)$.

5. If $E \equiv \text{sameTerm}(E_1, E_2)$,

$$[\![E]\!] \triangleq \begin{cases} \mathrm{RDF}(\text{true}) & \text{if } [\![E_1]\!] = [\![E_2]\!] \\ \text{error} & \text{if } [\![E_1]\!] = \text{error} \vee [\![E_2]\!] = \text{error} \\ \mathrm{RDF}(\text{false}) & \text{otherwise.} \end{cases}$$

6. If $E \equiv (E_1 = E_2)$,

$$[\![E]\!] \triangleq \begin{cases} \text{RDF(true)} & \text{if } [\![E_1]\!] = [\![E_2]\!] \\ \text{RDF}\Big(\text{value}([\![E_1]\!]) = \text{value}([\![E_2]\!])\Big) & \text{if } ([\![E_1]\!], [\![E_2]\!]) \in \\ & \quad \mathbb{L}^2_{ts} \cup \mathbb{L}^2_{tb} \cup \mathbb{L}^2_{tn} \cup \mathbb{L}^2_{td} \\ \text{RDF}\Big(\text{str}([\![E_1]\!]) = \text{str}([\![E_2]\!])\Big) & \text{if } ([\![E_1]\!], [\![E_2]\!]) \in \mathbb{L}^2_{ps} \\ \text{RDF(false)} & \text{if } ([\![E_1]\!], [\![E_2]\!]) \notin \mathbb{L}^2 \wedge \\ & \quad [\![E_1]\!] \neq [\![E_2]\!] \\ \text{error} & \text{otherwise.} \end{cases}$$

7. If $E \equiv (E_1 \neq E_2)$, $[\![E]\!] = [\![\neg(E_1 = E_2)]\!]$.

8. If $E \equiv (E_1 \bullet E_2)$, where $\bullet \in \{ <, \leqslant, >, \geqslant \}$,

$$[\![E]\!] \triangleq \begin{cases} \text{RDF}\Big(\text{value}([\![E_1]\!]) \bullet \text{value}([\![E_2]\!])\Big) & \text{if } ([\![E_1]\!], [\![E_2]\!]) \in \\ & \quad \mathbb{L}^2_{ts} \cup \mathbb{L}^2_{tb} \cup \mathbb{L}^2_{tn} \cup \mathbb{L}^2_{td} \\ \text{RDF}\Big(\text{str}([\![E_1]\!]) \bullet \text{str}([\![E_2]\!])\Big) & \text{if } ([\![E_1]\!], [\![E_2]\!]) \in \mathbb{L}^2_{ps} \\ \text{error} & \text{otherwise.} \end{cases}$$

9. If $E \equiv (E_1 \bullet E_2)$, where $\bullet \in \{ *, /, +, - \}$,

$$[\![E]\!] \triangleq \begin{cases} \text{RDF}\Big(\text{value}([\![E_1]\!]) \bullet \text{value}([\![E_2]\!])\Big) & \text{if } ([\![E_1]\!], [\![E_2]\!]) \in \mathbb{L}^2_{tn} \\ \text{error} & \text{otherwise.} \end{cases}$$

10. If $E \equiv \text{bound}(x)$, $[\![E]\!] \triangleq \text{false}$.

11. If $E \equiv f(E')$, where $f = \text{isIRI}$ (resp. isBlank and isLiteral),

$$[\![E]\!] \triangleq \begin{cases} \text{RDF(true)} & \text{if } [\![E']\!] \in \mathbb{I} \text{ (resp. } \mathbb{B} \text{ and } \mathbb{L}) \\ \text{error} & \text{if } [\![E']\!] = \text{error} \\ \text{RDF(false)} & \text{otherwise.} \end{cases}$$

12. If $E \equiv \text{str}(E')$, $[\![E]\!] \triangleq \begin{cases} (\text{str}([\![E']\!]), \text{""}) & \text{if } [\![E']\!] \in \mathbb{I} \cup \mathbb{L} \\ \text{error} & \text{otherwise.} \end{cases}$

13. If $E \equiv \text{lang}(E')$, $[\![E]\!] \triangleq \begin{cases} (\text{lang}([\![E']\!]), \text{""}) & \text{if } [\![E']\!] \in \mathbb{L}_p \\ (\text{""}, \text{""}) & \text{if } [\![E']\!] \in \mathbb{L}_t \\ \text{error} & \text{otherwise.} \end{cases}$

14. If $E \equiv \text{datatype}(E')$, $[\![E]\!] \triangleq \begin{cases} \text{datatype}([\![E']\!]) & \text{if } [\![E']\!] \in \mathbb{L}_t \\ \texttt{xsd:string} & \text{if } [\![E']\!] \in \mathbb{L}_{ps} \\ \text{error} & \text{otherwise.} \end{cases}$

Note that the SPARQL specification distinguishes between identity and equivalence. The $(\mathsf{sameTerm}(E_1, E_2))$ predicate asserts the identity of E_1 and E_2, i.e., whether they refer to the exact same RDF term. The $(E_1 = E_2)$ predicate asserts the equivalence of E_1 and E_2, i.e., whether their interpreted values are equal. The notion of equivalence is only defined on pairs of literals of the same type that SPARQL understands, i.e., simple literals, strings, booleans, numbers and dates. If E_1 and E_2 do not have the same type, or they have a type that is not understood by SPARQL, they are said to be incomparable. Plain literals with non-empty language tags are also incomparable. In such cases, $[\![E_1 = E_2]\!]$ falls back on the identity. If the values of E_1 and E_2 are not identical, the result is false, except if both operands are literals, in which case the result is error.

Similarly to expressions, we define the substitution of a solution mapping in a graph pattern. The evaluation of a graph pattern P over a graph G is denoted by $[\![P]\!]_G$.

Definition 2.12. The *substitution* of a solution mapping μ in a graph pattern P, denoted by $\mu[P]$, is the graph pattern obtained by applying the following operations on P.

1. Replace any variable $x \in \mathrm{dom}(\mu)$ occurring in triple patterns appearing in P by the value $\mu(x)$.
2. Replace any expression E appearing in P by the substitution $\mu[E]$.

Definition 2.13. The *evaluation* of a graph pattern P over a graph G, denoted by $[\![P]\!]_G$, is a set of solution mappings recursively defined as follows.

1. If P is a basic graph pattern, $[\![P]\!]_G \triangleq \{\, \mu \mid \mathrm{dom}(\mu) = \mathrm{vars}(P) \land \mu[P] \subseteq G \,\}$.
2. If $P \equiv (P_1 \text{ AND } P_2)$, $[\![P]\!]_G \triangleq \{\, \mu_1 \cup \mu_2 \mid \mu_1 \in [\![P_1]\!]_G \land \mu_2 \in [\![P_2]\!]_G \land \mu_1 \sim \mu_2 \,\}$.
3. If $P \equiv (P_1 \text{ UNION } P_2)$, $[\![P]\!]_G \triangleq \{\, \mu \mid \mu \in [\![P_1]\!]_G \lor \mu \in [\![P_2]\!]_G \,\}$.
4. If $P \equiv (P_1 \text{ DIFF } P_2)$, $[\![P]\!]_G \triangleq \{\, \mu_1 \mid \mu_1 \in [\![P_1]\!]_G \land \neg\exists \mu_2 \in [\![P_2]\!]_G , \mu_1 \sim \mu_2 \,\}$.
5. If $P \equiv (P_1 \text{ OPT } P_2)$, $[\![P]\!]_G \triangleq [\![(P_1 \text{ AND } P_2) \text{ UNION } (P_1 \text{ DIFF } P_2)]\!]_G$.
6. If $P \equiv (P' \text{ FILTER } E)$, $[\![P]\!]_G \triangleq \{\, \mu \mid \mu \in [\![P']\!]_G \land \mathrm{EBV}([\![\mu[E]]\!]) = \mathsf{true} \,\}$.

Evaluating a basic graph pattern involves finding a matching subset of the RDF graph. Such definition assumes no entailment regime is used. When using an entailment regime such as RDF Schema or OWL, evaluating a BGP amounts to finding an instance of the pattern graph that is entailed by the target RDF graph. Formally, the definition becomes $[\![P]\!]_G \triangleq \{\, \mu \mid \mathrm{dom}(\mu) = \mathrm{vars}(P) \land G \models \mu[P] \,\}$. If G is the deductive closure of the target RDF graph, the definitions are equivalent.

Example 2.3. Consider the target RDF graph G depicted in fig. 2.2, and the BGP $\{(p, \texttt{rdf:type}, \texttt{foaf:Person})\}$, where p is a variable. Under the simple entailment regime, the evaluation of the BGP yields no solution. When using the RDF Schema entailment regime with the rules depicted in fig. 2.4, the results are `phd:Tajel`, `phd:Cecilia`, `phd:Mike`, and `phd:Smith`. Indeed, we infer they are persons as they are involved in `foaf:knows` relations.

The AND operator is the concatenation of two patterns. The UNION operator produces the union of the solution sets of the operand patterns. The DIFF operator returns all the solutions of the left-hand operand that cannot be extended with a solution of the right-hand operand. Intuitively, $(P_1 \text{ OPT } P_2)$ *tries* to extend solutions of P_1 with solutions of P_2. However, if the extension of a solution $\mu_1 \in P_1$ fails (i.e., $\mu_1 \in [\![P_1 \text{ DIFF } P_2]\!]_G$), that solution μ_1 becomes a solution of the OPT pattern too. Figure 2.7 shows examples of the evaluation of compound patterns. The FILTER pattern only keeps the solutions of the subpattern for which the condition expression is satisfied.

Corollary. *Any solution mapping in the evaluation of a pattern P does not cover more variables than appear in P, i.e.,* $\mathrm{dom}(\mu) \subseteq \mathrm{vars}(P)$ *for all* $\mu \in [\![P]\!]_G$.

By construction, the domain of a solution of a basic graph pattern is the set of variables appearing in the pattern. The evaluation of a compound pattern combines the evaluation of the subpatterns without adding new variables.

The set of solution modifiers of a query is transformed into a modifier function. That function is applied on the evaluation of the graph pattern and returns the (modified) solutions in a sequence. The modifier function is composed of the Sort, Project, FilterDups and Slice functions.

The $\mathrm{Sort}(\langle O\rangle, \Omega)$ function returns all the elements of the set Ω in a sequence ordered by $\langle O\rangle$ lexicographically. For every ordering criterion (E, D) in $\langle O\rangle$, the sort key is given by $[\![\mu[E]]\!]$ with $\mu \in \Omega$. If $D = \text{ASC}$, the order direction is ascending. If $D = \text{DESC}$, the direction is descending.

The $\mathrm{Project}(X, \langle \mu_1, \ldots, \mu_n\rangle)$ function returns a sequence $\langle \mu'_1, \ldots, \mu'_n\rangle$, where $\forall i \in \{1, \ldots, n\}, \mathrm{dom}(\mu'_i) = \mathrm{dom}(\mu_i) \cap X$ and $\forall i \in \{1, \ldots, n\}, x \in \mathrm{dom}(\mu'_i), \mu'_i(x) = \mu_i(x)$.

The $\mathrm{FilterDups}(\langle \mu_1, \ldots, \mu_n\rangle)$ function returns the input sequence without duplicate elements. If there are duplicate elements at different positions of the input sequence, it is not specified which element to keep.

The $\mathrm{Slice}(n_O, n_L, \langle \mu_1, \ldots, \mu_n\rangle)$ function returns the sequence $\langle \mu_{n_O+1}, \ldots, \mu_{n_O+n_L}\rangle$.

Definition 2.14. The *modifier function m*, mapping a set of solutions Ω to a sequence of solutions, associated with a set of solution modifiers M is

$$m(\Omega) = \begin{cases} \mathrm{Slice}(n_O, n_L, \mathrm{FilterDups}(\mathrm{Project}(X, \mathrm{Sort}(\langle O\rangle, \Omega)))) & \text{if DISTINCT} \in M \\ \mathrm{Slice}(n_O, n_L, \mathrm{Project}(X, \mathrm{Sort}(\langle O\rangle, \Omega))) & \text{otherwise,} \end{cases}$$

where $n_O = n$ if $\text{OFFSET}(n) \in M$ or 0 otherwise, $n_L = n$ if $\text{LIMIT}(n) \in M$ or ∞ otherwise, X is the set given by $\text{PROJECT}(X)$ if $\text{PROJECT}(X) \in M$ or $\mathbb{V}$ otherwise, $\langle O\rangle$ is the sequence given by $\text{ORDER}(\langle O\rangle)$ if $\text{ORDER}(\langle O\rangle) \in M$ or the empty sequence otherwise.

Definition 2.15. The *evaluation* of a query instance (Q, G), with $Q = (P, M)$, is the sequence of solution mappings $m([\![P]\!]_G)$, where m is the modifier function associated to M.

P_1

p	a
phd:Cecilia	26
phd:Mike	35
phd:Tajel	29

P_2

p	t
phd:Cecilia	dbp:Procrastination
phd:Mike	dbp:Procrastination
phd:Mike	phd:Free Food
phd:Smith	dbp:Research

P_1 AND P_2

p	a	t
phd:Cecilia	26	dbp:Procrastination
phd:Mike	35	dbp:Procrastination
phd:Mike	35	phd:Free Food

P_1 UNION P_2

p	a	t
phd:Cecilia	26	
phd:Mike	35	
phd:Tajel	29	
phd:Cecilia		dbp:Procrastination
phd:Mike		dbp:Procrastination
phd:Mike		phd:Free Food
phd:Smith		dbp:Research

P_1 DIFF P_2

p	a
phd:Tajel	29

P_1 OPT P_2

p	a	t
phd:Cecilia	26	dbp:Procrastination
phd:Mike	35	dbp:Procrastination
phd:Mike	35	phd:Free Food
phd:Tajel	29	

Figure 2.7: Examples of pattern evaluation on the example graph of fig. 2.2 for combinations of the BGPs $P_1 \equiv \{(p, \texttt{foaf:member}, \texttt{phd:Students}), (p, \texttt{foaf:age}, a)\}$ and $P_2 \equiv \{(p, \texttt{foaf:interest}, t)\}$. Each row is a solution. A blank cell indicates that the variable is not present in the solution.

Example 2.4. Consider the solution set Ω of the graph pattern P_1 OPT P_2 in fig. 2.7, and the solution modifiers

$$M = \left\{ \text{PROJECT}(\{p, a\}), \text{DISTINCT}, \text{ORDER}(\langle(a, \text{ASC})\rangle), \text{LIMIT}(2), \text{OFFSET}(1) \right\} .$$

The associated modifier function is

$$m(\Omega) = \text{Slice}(1, 2, \text{FilterDups}(\text{Project}(\{p, a\}, \text{Sort}(\langle(a, \text{ASC})\rangle, \Omega)))) .$$

Figure 2.8 shows the intermediate result for each function.

$\Omega_0 = [\![P]\!]_G$

p	a	t
phd:Cecilia	26	dbp:Procrastination
phd:Mike	35	dbp:Procrastination
phd:Mike	35	phd:Free Food
phd:Tajel	29	

$\Omega_1 = \text{Sort}(\langle(a, \text{ASC})\rangle, \Omega_0)$

p	a	t
phd:Cecilia	26	dbp:Procrastination
phd:Tajel	29	
phd:Mike	35	dbp:Procrastination
phd:Mike	35	phd:Free Food

$\Omega_2 = \text{Project}(\{p, a\}, \Omega_1)$

p	a
phd:Cecilia	26
phd:Tajel	29
phd:Mike	35
phd:Mike	35

$\Omega_3 = \text{FilterDups}(\Omega_2)$

p	a
phd:Cecilia	26
phd:Tajel	29
phd:Mike	35

$\Omega_4 = \text{Slice}(1, 2, \Omega_3)$

p	a
phd:Tajel	29
phd:Mike	35

Figure 2.8: The solution set of the graph pattern is transformed by the modifier function into the solution sequence of the query. The set Ω_0 is the evaluation result of the P_1 OPT P_2 pattern from fig. 2.7. The sequence Ω_4 is the result of the query.

Chapter 3

Relational Databases Technology used in SPARQL Engines

State-of-the-art SPARQL engines rely on relational database technology. This chapter will explain how such engines store RDF graphs and how they evaluate SPARQL queries. As a running example, we will consider the dataset shown in the previous chapter in fig. 2.2, and the query of fig. 3.1. This chapter is based on surveys by Sakr and Al-Naymat [SA10], Hose et al. [Hos+11], and Luo et al. [Luo+12].

The first section will provide a broad overview of state-of-the-art SPARQL engines. The following sections will dive into the details of data storage (section 3.2) and query processing (section 3.3).

3.1 Overview of State-of-the-art SPARQL Engines

State-of-the-art SPARQL engines based on relational database technology store the triples of an RDF graph in relational tables. A SPARQL query can then be seen as an SQL query over those tables. The SQL query is evaluated using standard techniques.

Hose et al. [Hos+11] divides state-of-the-art systems in three classes, based on the structure of the relational tables:

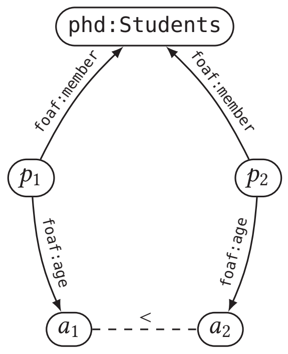
1. *triple stores* that store the whole dataset in one three-column table,
2. *vertically partitioned tables* that maintain one table for each predicate, and
3. *property tables* where several predicates are jointly represented.

This section gives an overview of each class.

Because of their popularity, the next sections will focus on triple stores. Vertically partitioned tables have few advantages over triple stores with efficient indexes. Property tables are a middle-ground between the open nature of the semantic web and the structured data of relational databases.

```
SELECT ?p1 ?p2 WHERE {
  ?p1 foaf:member phd:Students .   (t1)
  ?p1 foaf:age ?a1 .               (t2)
  ?p2 foaf:member phd:Students .   (t3)
  ?p2 foaf:age ?a2 .               (t4)
  FILTER (?a1 < ?a2)               (E)
}
```

(a) SPARQL Query



(b) Associated BGP

Figure 3.1: The example query finds all pairs of PhD students where the first one is strictly younger than the second one. Such inequality filters are common to break symmetries.

subject	predicate	object
phd:Students	rdf:type	foaf:group
phd:Tajel	foaf:member	phd:Students
phd:Tajel	foaf:name	("Tajel", "")
phd:Tajel	foaf:age	("29",xsd:integer)
phd:Tajel	foaf:knows	phd:Cecilia
phd:Tajel	foaf:knows	phd:Mike
phd:Cecilia	foaf:member	phd:Students
phd:Cecilia	foaf:name	("Cecilia",xsd:string)
...	...	...

Figure 3.2: A triple store stores all triples in one single triple table. To save space, terms are usually mapped to identifiers (not shown here).

3.1.1 Triple Stores

Triple stores store the whole dataset in one giant table, called the *triple table*. Each row in the table represents one triple. Figure 3.2 shows a part of the triple table for the dataset of our running example (see fig. 2.2). Examples of SPARQL engines which use triple stores are Sesame [BKH02], 4store [HLS09], Virtuoso [EM09], RDF-3X [NW08], and Hexastore [WKB08].

To save space and improve efficiency, most systems assign an integer identifier to every RDF term appearing in the dataset, e.g., with a hashing function or with consecutive integers. The triple table then contains only the identifiers, making it more compact. Of course, the mapping has to be stored in an additional table.

Some systems, e.g., Sesame, 4store or Virtuoso, are able to store multiple RDF graphs at once. The relational table is extended with a fourth column, identifying for each triple the provenance graph by its IRI. Such extended table is called a *quadruple table*. In this thesis, we will focus on triple tables. In most cases, the extension to quadruple tables is straightforward.

The basic query operation on a triple store is to retrieve all triples matching a triple pattern. Remember a triple pattern is a triple where each component is either a constant term or a variable, meaning any term can appear at that place. To answer such queries efficiently, indexes are maintained on (a subset of) all combinations of columns of the triple table. The size of an index is approximately as large as the triple table. Thus, the number of indexes maintained in a system is usually limited, or the index has to be stored in a compressed format. Section 3.2 will provide more details.

A SPARQL query can be translated to an SQL query on the triple table. For each triple pattern of the query, a copy of the triple table is included in the query. Whenever a common variable is used in two triple patterns, a join is introduced between the two corresponding table instances on the columns where the variable occurs. A condition is added for every constant. Filters are translated to an equivalent SQL condition.

Example 3.1. The query in fig. 3.1 is translated to the following SQL query.

```sql
SELECT t1.s, t3.s
FROM triples t1, triples t2, triples t3, triples t4
WHERE t1.p = 'foaf:member'
  AND t1.o = 'phd:Students'
  AND t2.p = 'foaf:age'
  AND t3.p = 'foaf:member'
  AND t3.o = 'phd:Students'
  AND t4.p = 'foaf:age'
  AND t1.s = t2.s
  AND t3.s = t4.s
  AND t2.o < t4.o
```

A copy of the triple table, named `triples`, is included for each triple pattern. Four copies are included, named `t1`, `t2`, `t3`, and `t4`. Conditions are stated on the subject (`s`), predicate (`p`), and object (`o`) columns. In a real system, the constants would be first mapped to the corresponding identifiers. Similarly, the results would be translated back to RDF terms.

The obtained SQL query can be evaluated using standard relational database techniques. This involves converting the query in an equivalent tree of abstract operators, i.e., join, projection, and selection operators. The abstract operators are then mapped to physical operators that are executed. Key choices are the ordering of the operators and the choice of the physical operators. Standard heuristics for relational databases rely on statistics on the columns of the tables. However, such statistics do not provide enough information when applied on the single triple table. Specific heuristics are thus needed for SPARQL processing. Such heuristics, along with a more detailed description of the query processing, are described in section 3.3.

`foaf:member`

subject	object
`phd:Tajel`	`phd:Students`
`phd:Cecilia`	`phd:Students`
`phd:Mike`	`phd:Students`

`foaf:age`

subject	object
`phd:Tajel`	("29",`xsd:integer`)
`phd:Cecilia`	("26",`xsd:integer`)
`phd:Mike`	("35",`xsd:decimal`)
`phd:Smith`	("56",`xsd:integer`)

`foaf:name`

subject	object
`phd:Tajel`	("Tajel","")
`phd:Cecilia`	("Cecilia",`xsd:string`)
`phd:Mike`	("Michael Slackenerny","")
`phd:Smith`	("Brian B. Smith","")

...

Figure 3.3: A system with vertically partitioned tables groups the triples by predicate (not all tables are shown). The evaluation of triple patterns with constant predicate involves much smaller tables than in a triple store. As for triple stores, terms are usually mapped to identifiers (not shown here).

3.1.2 Vertically Partitioned Tables

In most real-world SPARQL queries, predicates are constant. Vertically partitioned tables exploit this property. A two-column table is created for each predicate p, containing the subject-object pairs of the triples with predicate p. Figure 3.3 shows how the RDF graph of fig. 2.2 is stored in such scheme. SW-Store [Aba+09] is an example of vertically partitioned system.

Instead of storing the tables in a traditional relational database, i.e., a *row store*, one can also rely on a *column store*. A row store considers a table as a collection of rows. A column store stores a table as a collection of columns. As all data within a column have the same type, such columns can be compressed efficiently. MonetDB [Idr+12] is the prime example of column store. Recently, Virtuoso has added support for column-wise tables [Erl12].

Vertically partitioned tables have two main advantages with respect to query processing. Triple patterns with constant predicate can be evaluated efficiently by scanning the table of the predicate, which is much smaller than the triple table of triple stores. However, such advantage is limited when using efficient indexes in triple stores (see section 3.2). Statistics on the vertically partitioned tables give more accurate estimations when using standard heuristics. Specialized heuristics are thus less needed. On the other hand, queries involving variable predicates are very expensive to compute because they need to iterate over all two-column tables and return the union of the results.

Example 3.2. The query in fig. 3.1 is translated to the following SQL query.

```
SELECT t1.s, t3.s
```

```
FROM ‘foaf:member‘ t1, ‘foaf:age‘ t2, ‘foaf:member‘ t3, ‘foaf:age‘ t4
WHERE t1.o = ’phd:Students’
  AND t3.o = ’phd:Students’
  AND t1.s = t2.s
  AND t3.s = t4.s
  AND t2.o < t4.o
```

3.1.3 Property Tables

While RDF does not require any structure for the data, most datasets have an implicit structure. Many resources appearing as subjects in the dataset can be partitioned in a set of classes. Subjects in a class share the same, or a largely overlapping, set of properties. For example, in fig. 2.2 every person has a name and an age. Property tables group together all the properties of a subject in one row. Jena [Car+04] is an example of such system.

A row in a property table represents a set of triples with the same subject. The first column is the subject s. The other columns represent various predicates. The value of a column p is the object p if the triple (s,p,o) exists in the dataset or NULL otherwise. Figure 3.4 shows the property tables of the people and the web pages of fig. 2.2.

Queries often access multiple properties of a subject. This is the case in our running example of fig. 3.1, where we access the group and the age of each person. Property tables are able to handle queries more efficiently by avoiding to join tables to combine multiple properties. For this reason, property tables are able to outperform triple stores and vertically partitioned tables on very structured datasets [LM09].

Example 3.3. The query in fig. 3.1 is translated to the following SQL query.

```
SELECT p1.subject, p2.subject
FROM Person p1, Person p2
WHERE p1.‘foaf:member‘ = ’phd:Students’
  AND p2.‘foaf:member‘ = ’phd:Students’
  AND p1.‘foaf:age‘ < p2.‘foaf:age‘
```

The above query is very similar to what one may obtain when using standard relational databases instead of RDF.

Property tables have a number of limitations however. Because RDF has a schemaless nature, the structure of the data needs to be (re)discovered. If the user provides a schema, e.g., with RDF-Schema or OWL, such information can be used. Otherwise, heuristics are needed. There should not be too many NULLs in the tables as they increase the storage space. Thus, storing the whole dataset in one table is not an option. Instead, the triples need to be clustered heuristically in some way.

Another problem comes with multi-valued properties, i.e., triples with the same subject and predicate, but different objects. In our example, this happens with the

Person

subject	foaf:member	foaf:name	foaf:age	foaf:weblog
phd:Tajel	phd:Students	"Tajel"	29	NULL
phd:Cecilia	phd:Students	"Cecilia"	26	http://...
phd:Mike	phd:Students	"Mike"	35	NULL
phd:Smith	NULL	"Brian B. Smith"	56	NULL

Web page

subject	foaf:topic	dc:created	dc:subject
http://...	dbp:Procrastination	2005-07-10	_:a

⋮

Remainder

subject	predicate	object
phd:Tajel	foaf:knows	phd:Cecilia
phd:Tajel	foaf:knows	phd:Mike
phd:Cecilia	foaf:knows	phd:Tajel
phd:Mike	foaf:knows	phd:Smith
...	...	...

Figure 3.4: Property tables group together the properties of a subject in a single row (not all tables are shown). Such tables reflect the underlying structure of the data. They are very close to the tables of a standard relational database. To fit on the page, the datatypes of the literals are omitted.

`foaf:knows` predicate. One solution would be to duplicate the columns to accommodate for multiple values. However, this can only be done if we know the maximum number of values in advance. The other solution is to resort to a triple table to store the remainder triples that cannot be expressed inside property tables. Such solution is shown in fig. 3.4.

3.2 Triple Indexes

The simplest queries consist of a single triple pattern. Such queries are the basis of more complex queries. To answer them efficiently, triple stores make use of indexes. An index is an auxiliary data structure that helps to efficiently retrieve triples satisfying some conditions, e.g., all triples whose predicate is `foaf:member` and whose object is `phd:Students`. Note that our definition of index is intentionally large. Not all indexes return full triples. For example, an index could return only the subjects that appear in triples whose predicates are `foaf:member`.

In this section, we will first introduce the underlying data structures. Then, we will show how they are used in state-of-the-art triple stores.

3.2.1 Data Structures

Conceptually, the data structures used for indexes are maps. They map *keys* (e.g., a predicate) to *values* (e.g., the triples with the given predicate). In the context of databases, a value is also called a *payload*. Maps can also be used as mathematical sets by using an empty payload. Table 3.1 shows an overview of the data structures presented in this section.

B-trees

B-trees [Com79] are the most common data structures used for indexing in relational databases. The complexity of a look-up is $O(\log n)$, with n the number of indexed keys. B-trees are designed to work well when the cost of reading a node is high, e.g., it has to be read from the hard disk. We will discuss a particular variant, B$^+$-trees, that often occur in relational databases.

	B-tree	Radix trie	Hash table	Bitmap
Use	Map or set	Map or set	Map or set	Set
Look-up complexity	$O(\log n)$	$O(k)$	$O(1)$	$O(1)$
In-order traversal	Yes	Yes	No	Yes

Table 3.1: Indexes can be implemented with various data structures having different strengths and weaknesses.

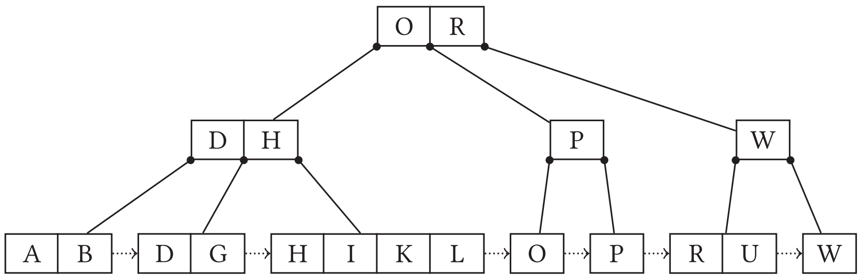


Figure 3.5: In a B$^+$-tree, the key-value pairs are stored in the leaf nodes, i.e., the bottom level (only the keys are shown here). Internal nodes have multiple keys. Leaf nodes are usually linked together to allow efficient in-order traversal (shown with dotted arrows).

A B$^+$-tree distinguishes between internal nodes and leaf nodes. Leaf nodes contain consecutive keys along with their values. Internal nodes contain only keys. An internal node with k keys has $k + 1$ children, partitioning the keys of its children. For example, an internal node with keys 'O' and 'R' will have three child subtrees. The first one will have all keys smaller than 'O', the second one keys between 'O' (inclusive) and 'R' (exclusive), and a third one with keys greater than or equal to 'R'. To provide efficient in-order traversal, leaf nodes are usually linked together. Figure 3.5 shows an example of a B$^+$-tree.

In a B$^+$-tree, all nodes take the same amount of space on disk, called a *page*. The time to read a page includes access time, i.e., the time to find the page on the disk, and read time, i.e., the time to actually read the content of the page. On traditional hard disks, the access time is not negligible, and the page size is a trade-off between access time and read time. Typical page sizes are 8 or 16 KB. The number of keys stored in a node depends on the compression scheme inside the pages.

To ensure good look-up performances, the tree has to be balanced. Updating the tree, i.e., inserting or removing keys, may involve splitting or merging nodes up to the root in order to keep the balanced property. Because such operations can be costly, engines sometimes resort to tricks to reduce the number of balancing operations that have to be performed. One trick is to leave some empty space in all nodes to accommodate for small insertions [Erl12]. Another trick is to maintain a small delta structure that is periodically merged into the B$^+$-tree [Hém+08; NW10].

Radix tries

A radix trie, also known as a Patricia tree [Mor68], is a prefix tree where a node with a single child is merged with its child. The worst-case complexity of a look-up is $O(k)$, with k the size of the keys. This is worse than the complexity of B-trees since $k \geqslant \log n$ (we need at least $\log n$ bits to distinguish between n elements). On the other hand, updates to the data structure do not need costly balancing operations.

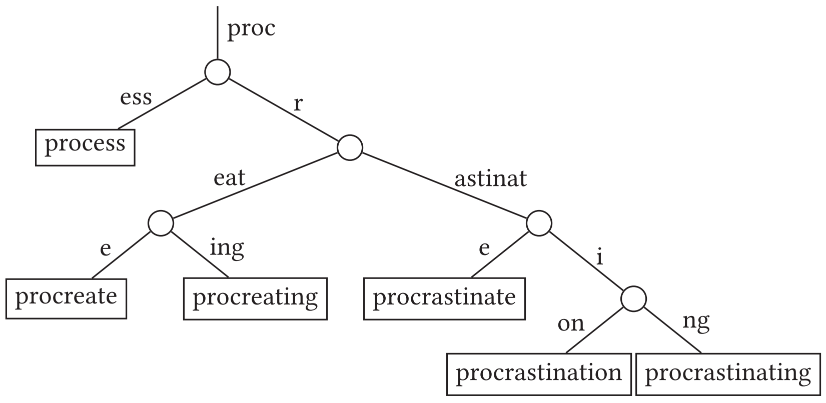


Figure 3.6: A radix trie is a prefix tree where edge labels may be sequences of elements, i.e., strings, instead of a single element. Values are attached to the leaf nodes (not shown here).

In a radix trie, edges are labeled with sequences of elements. An element is a part of a key, e.g., a character, a bit, a byte, etc. The key corresponding to a leaf node can be reconstructed by concatenating the labels on the path to the leaf. Internal nodes are empty. The leaves contain the payloads. Figure 3.6 shows an example of a radix trie.

Hash tables

In a hash table, a key is transformed to a table index by applying successively a hashing and a compression function. That index points to an entry of the hash table. A collision occurs when two different keys map to the same index. An entry can be a list, also known as a bucket, of key-value pairs with all colliding keys. Alternatively, an entry can also be a single key-value pair. A new key having the same index will then be stored in the nearest empty entry. Such method is called open addressing.

The average time complexity of a look-up is $O(1)$ if there are not too many collisions. When inserting new keys, the table sometimes has to be extended. The extension is a very costly operation as all items have to be rehashed and moved to the new index.

Bitmaps

Bitmaps can only represent mathematical sets, i.e., maps with no payloads. A bitmap consists of a bit array, also called a bit vector. A key is a natural number, representing the index of a bit in the bit vector. The set contains the key if and only if its associated bit is 1. Such look-up is performed in constant time.

A big advantage of bitmaps is the ability to perform bit-wise operations. For example joining two bitmaps, i.e., computing the intersection of two sets, involves a bit-wise AND operation. Computers perform such operations very efficiently.

3.2.2 Mapping RDF Terms to Identifiers

Before diving into the details of how triple stores store their indexes, we will briefly explain how RDF terms are mapped to integer identifiers. Handling identifiers has two major advantages over dealing with terms directly. First, identifiers have fixed length in contrast to the variable length of RDF terms. More efficient fixed-length records can thus be used. Second, IRIs often appear multiple times in an RDF graph. Storing them only once and replacing them with shorter identifiers results in significant space savings. There are two general approaches to the problem: hash-based and counter-based.

Hash-based approaches apply a hash function to the RDF terms. For example, 4store [HLS09] uses a 64-bit key to identify the terms. Care must be taken to distinguish between the various types of RDF terms, e.g., IRIs and literals. In 4store, such information is encoded in the most significant bits. Hash collisions must be handled to avoid wrong query results. Because collisions are unlikely, 4store refuses to load a triple if it detects a collision.

Counter-based approaches simply assign consecutive integers to RDF terms. When a new term is encountered, a counter is incremented and the value of the counter is the new identifier. Such approach is chosen by RDF-3X [NW08].

To translate identifiers back to their original term, a *dictionary* table is needed. The dictionary can use various data structures. For example, 4store uses a hash table, and RDF-3X uses a B^+-tree.

Variations on the general approaches are possible. For instance, Virtuoso [Erl12] uses small terms (up to 8 bytes) directly as identifiers. Common prefixes of IRIs can also be compressed more efficiently.

3.2.3 Indexes in Triple Stores

State-of-the-art triple stores have different approaches to index the triple table. In this section, we will give an overview of the indexes used by Virtuoso, 4store, and RDF-3X.

As noted above, an index is basically a key-value map. We will denote keys by combinations of the following letters: S (subject), P (predicate), O (object), and G (graph IRI). For example, the keys of an SP index are pairs of subjects and predicates. In a B-tree, the order of the components specify the lexical ordering in the tree. For example, keys in an SP index will first be ordered by subject, then by predicate.

We distinguish between full and partial indexes. In a *full* index, it is possible to reconstruct full triples with the keys and their payloads. A full index is effectively a complete copy of the triple table, albeit ordered differently. On the other hand, a *partial* index alone does not allow to reconstruct the triple table.

Virtuoso

Virtuoso [Erl12] is very flexible and allows the database administrator to specify the indexes to create. The default indexing scheme includes 2 full indexes and 3 partial indexes.

The first full index is a B-tree with PSOG keys and no payload. This index is used to answer triple patterns with constant predicate, and possibly constant subject. To handle patterns with constant predicate and object, the second full index is a B-tree with POG keys and S bitmaps as payloads. Hence, for every combination of predicate, object, and graph IRI appearing in the dataset, a bitmap containing the associated subjects is stored.

To answer queries with variable predicates or constant graph IRI, the OP, SP, and GS partial indexes are introduced. E.g., the OP index maps an object to predicates. The full PSOG index can then be queried with the resulting predicates to retrieve the full triples. The OP index is a B-tree with OP keys and no payloads. The SP (resp. GS) index is a B-tree with S (resp. G) keys and P (resp. S) bitmap payloads.

4store

Even though 4store [HLS09] is considered as a triple store, its index structure resembles vertically partitioned tables. For each predicate, two radix tries are built, one with the subjects as keys, the other with predicates. The payloads contain lists of triples matching the predicate and subject/object. Because the indexes are full indexes, the triple table is not stored on disk.

A hash table maps graph IRIs to lists of triples. This full index allows to efficiently retrieve the triples of one graph. As a side effect, it also allows to quickly delete a whole graph.

The authors of 4store have chosen radix tries over B-trees for their easy insertions without costly balancing operations. Because identifiers are evenly distributed thanks to the hashing function, the worst-case conditions of radix tries should be uncommon.

RDF-3X

In contrast to the previous engines, RDF-3X does not support multiple graphs. The triple table is not materialized. Instead, full B^+-tree indexes with no payloads are maintained for all six permutations of the columns: SPO, SOP, PSO, POS, OSP, and OPS. While three indexes are enough to handle any triple pattern, the additional indexes allow for different orderings of the results. For example, for a triple pattern with constant predicate, the PSO index will return triples ordered first by subject, then by object. The POS index will instead return triples ordered first by object, then by subject. The order of the results have an impact on the join operators described in section 3.3.

RDF-3X also stores aggregated (partial) indexes for all possible pairs of columns: SP, SO, PS, PO, OS, and OP. A payload in one of those indexes consists of the number of triples with the components specified by the key. For example, the PO index of the graph shown in fig. 2.2 will return 3 for key (`foaf:member`,`phd:Student`), because there are three triples with that predicate and object. Similar aggregated indexes are also created for S, P, and O, returning the number of triples with the given subject, predicate, or object.

Aggregated indexes are used to efficiently answer queries such as `SELECT ?s ?o WHERE { ?s ?p ?o }`. Scanning through the SOP index would unnecessarily return all predicates for each SO pair. Using the SO index instead skips over the predicates. Aggregated indexes also give useful statistics to use during the query execution (see section 3.3).

To reduce the space requirements of the indexes, RDF-3X compresses the leaf pages of the B^+-trees. Because all identifiers are consecutive integers and the triples in a leaf are sorted, a delta-compression scheme is applied. The leaf starts with a full triple. For the next triples, only the difference with the previous triple is stored.

3.3 Query Execution and Optimization

Thanks to the indexes described in the previous section, triples matching a triple pattern can be efficiently retrieved. From each triple, a solution mapping can be constructed, assigning RDF terms to the variables. In order to evaluate a complete query, the result sets of the triple patterns must be combined. The first step in the query execution is to convert the SPARQL query into a tree of abstract operators. A physical operator, i.e., an implementation, is then chosen for each abstract operator, and the tree is executed bottom-up. The result set of the query is the result set of the root operator.

In this section, we will first describe the two steps of the query execution, i.e., the abstract and physical operators. Then, we will explain how state-of-the-art systems optimize the query execution.

3.3.1 Abstract Operators

An abstract operator transforms/combines the result sets of its operand(s) and returns a new result set. A SPARQL query is transformed in a tree of abstract operators. Such a tree is very similar to the input query as described in section 2.3.2. However, basic graph patterns are split into their triple patterns, which are combined with AND operators.

Figure 3.7 shows the abstract operator tree for the query of fig. 3.1. The leaves of the abstract operator tree are the triple patterns. The inner nodes are one of the following operators. The projection operator (π) restricts the domain of the solutions to a set of variables. The join ($\bowtie$), union ($\cup$), difference ($\setminus$), left-join ($\ltimes$), and selection (σ) operators correspond respectively to the SPARQL AND, UNION, DIFF, OPT, and FILTER operators.

By evaluating the abstract operator tree bottom-up, we obtain the solutions of the query. Figure 3.8 shows the intermediate result tables of the abstract operators when evaluating the tree of fig. 3.7 on the RDF graph of fig. 2.2. Depending on the physical operators chosen to implement the abstract operators (see section 3.3.2), some of the intermediate result tables need to be materialized, while others do not. Materializing an intermediate result involves computing the complete result set and storing it in memory or on disk for further processing by the parent operator.

Splitting a basic graph pattern into a tree of joined triple patterns can be done in various ways. Choosing the right tree has a great impact on the performances. For example, if we had chosen to first join t_1 and t_3, which have no common variable, the intermediate result tables would have been larger. Some ways of dealing with this problem are explained in section 3.3.3.

3.3.2 Physical Operators

Each abstract operator can be implemented in various ways. An implementation is called a physical operator. In the best case, an operator does not need to materialize the result tables of its operands. In such cases, the operator is applied on iterators that compute the solutions lazily. Hence, only one solution per operator needs to be kept in memory.

Some operators, such as the projection or the selection operators, are trivial to implement. In contrast, there exist a lots of physical operators implementing the join operator. The choice of physical operator depends on the operators lower in the tree and on the available indexes. We will describe the three fundamental physical join operators: nested loop join, hash join, and merge join. In what follows, we consider the join $A \bowtie B$ of two tables, such that variable x is shared between the two tables. One can easily generalize to multiple shared variables.

The *nested loop join* is the simplest implementation. For each solution $\mu_A \in A$, we iterate over the solutions $\mu_B \in B$. If the two solutions are compatible, $\mu_A \cup \mu_B$ is returned. The nested loop join requires one of the two result tables to be materialized, as we need to traverse it multiple times.

The *hash join* also requires one of the tables, e.g., B, to be materialized. We first build a hash table, mapping values for x to solutions of table B. Then, we traverse table A. For each solution μ_A, we look up $\mu_A(x)$ in the hash table. If we find a solution μ_B, we return $\mu_A \cup \mu_B$. Otherwise, no compatible solution exist and we advance to the next solution of A.

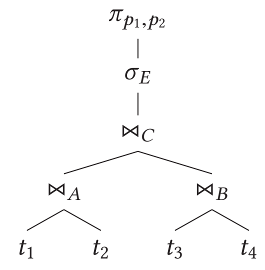
The *merge join* is the most efficient implementation. No table needs to be materialized. However, the tables A and B are required to be sorted by x. Let μ_A (resp. μ_B) be the current solution of the iterator on table A (resp. B). There are three cases:

1. If $\mu_A(x) = \mu_B(x)$, both solutions are compatible and we return $\mu_A \cup \mu_B$. The iterators on either A or B is advanced.

2. If $\mu_A(x) < \mu_B(x)$, we advance the iterator on A.

```
SELECT ?p1 ?p2 WHERE {
  ?p1 foaf:member phd:Students .    (t1)
  ?p1 foaf:age ?a1 .                (t2)
  ?p2 foaf:member phd:Students .    (t3)
  ?p2 foaf:age ?a2 .                (t4)
  FILTER (?a1 < ?a2)                (E)
}
```

(a) SPARQL query from fig. 3.1



(b) Abstract operator tree

Figure 3.7: A SPARQL query is converted to an abstract operator tree. Such a tree is similar to the SPARQL query with the exception of basic graph patterns, which are decomposed into triple patterns.

t_1

p_1
phd:Cecilia
phd:Mike
phd:Tajel

t_2

p_1	a_1
phd:Cecilia	26
phd:Tajel	29
phd:Mike	35
phd:Smith	56

t_3

p_2
phd:Cecilia
phd:Mike
phd:Tajel

t_4

p_2	a_2
phd:Cecilia	26
phd:Tajel	29
phd:Mike	35
phd:Smith	56

$\bowtie_A$

p_1	a_1
phd:Cecilia	26
phd:Mike	35
phd:Tajel	29

$\bowtie_B$

p_2	a_2
phd:Cecilia	26
phd:Mike	35
phd:Tajel	29

$\bowtie_C$

p_1	a_1	p_2	a_2
phd:Cecilia	26	phd:Cecilia	26
phd:Cecilia	26	phd:Mike	35
phd:Cecilia	26	phd:Tajel	29
phd:Mike	35	phd:Cecilia	26
phd:Mike	35	phd:Mike	35
phd:Mike	35	phd:Tajel	29
phd:Tajel	29	phd:Cecilia	26
phd:Tajel	29	phd:Mike	35
phd:Tajel	29	phd:Tajel	29

σ_E

p_1	a_1	p_2	a_2
phd:Cecilia	26	phd:Mike	35
phd:Cecilia	26	phd:Tajel	29
phd:Tajel	29	phd:Mike	35

π_{p_1,p_2}

p_1	p_2
phd:Cecilia	phd:Mike
phd:Cecilia	phd:Tajel
phd:Tajel	phd:Mike

Figure 3.8: The abstract operator tree in fig. 3.7 is evaluated bottom-up. The solutions of the query are the results of the root operator π_{p_1,p_2}. Because $\bowtie_A$ and $\bowtie_B$ do not share any variable, the result set of $\bowtie_C$ is the Cartesian product of both tables. Depending on the physical operators, the intermediate result tables are not always materialized. Note that the datatypes of literals have been omitted in the tables.

3. If $\mu_A(x) > \mu_B(x)$, we advance the iterator on B.

The algorithm is repeated until one of the two iterators reaches the end of the table.

3.3.3 Query Optimization

Building an efficient query plan, i.e., constructing the abstract operator tree and assigning physical operators, has a high impact on the query evaluation performances. Relational database systems resort to dynamic programming or randomized search to select the best plan [NW09]. Such techniques need to estimate the cost of a query plan. The cost of a query plan is directly related to the number of solutions generated by each operator, i.e., the selectivity of the operator.

A standard technique used in off-the-shelf relational databases involves attribute-level histograms. Such histograms represent the distribution of the values for each column of each table. However, the histograms ignore the correlation of columns. Therefore, the estimates are often wrong for the single triple table of our triple stores.

RDF-3X uses the counts stored in the aggregated indexes as better histograms. With these indexes, it can accurately predict the number of triples that a triple pattern will generate. The information is then used to infer estimates for the join operators.

Virtuoso relies on query-time sampling. For triple patterns with constant predicate and object, Virtuoso loads the first page of the bitmap in the POGS index. Based on this page, it extrapolates the number of results that the triple pattern would generate.

Many other optimization techniques exist for processing SPARQL queries, including algebraic rewriting (e.g., pushing filters down the operator tree) [SML10] and sideways information passing letting join operators communicate with each other [NW09].

Chapter 4

Constraint Programming

Constraint Programming (CP) is a programming paradigm designed to solve combinatorial NP-hard problems. CP has been shown to be efficient for graph matching problem [CDS09], which are closely related to SPARQL queries [Bag05].

The first section gives a general overview of CP. The following sections focus on essential aspects of the CP framework: variables (section 4.2), constraints (section 4.3), and search (section 4.4). Finally, section 4.5 presents the Comet solver, and section 4.6 gives a short overview of existing CP approaches for solving graph matching problems.

4.1 Overview of Constraint Programming

Constraint Programming is basically a technique to solve Constraint Satisfaction Problems (CSPs). A CSP is a declarative way to state a problem where one has to assign values to variables, such that a set of constraints are satisfied. The domain of each variable restricts the set of values that can be assigned to that variable. In this chapter, we consider finite domains and, without loss of generality, we assume that domains are sets of integer values.

Definition 4.1. A Constraint Satisfaction Problem (CSP) is a triple (X, D, C) where

- X is a set of variables,
- $D : X \rightarrow \mathcal{P}\mathbb{N}$ is a function mapping each variable to a domain, i.e., the finite set of integer values that can be assigned to the variable,
- C is a set of constraints on the variables of X.

Definition 4.2. A *constraint* c over a set of variables $\text{vars}(c) = \{x_1, \ldots, x_k\}$ is a mathematical relation $c \subset \mathbb{N}^k$. An assignment $\mu : \text{vars}(c) \rightarrow \mathbb{N}$ satisfies the constraint c if $(\mu(x_1), \ldots, \mu(x_k)) \in c$.

Definition 4.3. A *solution* of a CSP (X, D, C) is an assignment $\mu : X \to \mathbb{N}$ of all the variables in X, such that $\forall x \in X, \mu(x) \in D(x)$ and all the constraints in C are satisfied by μ.

Example 4.1. A Latin square is an $n \times n$ grid that contains numbers from 1 to n such that each number appears only once on any row and any column. Figure 4.1 shows an example. Finding a Latin square amounts to solving the CSP (X, D, C) such that

- $X = \{ x_{ij} \mid 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n \}$,
- $D(x) = \{ 1, \ldots, n \} \quad \forall x \in X$,
- $C = \{ x_{ij} \neq x_{ik} \mid 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n, j < k \leqslant n \}$
  $\quad \cup \{ x_{ij} \neq x_{kj} \mid 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n, i < k \leqslant n \}$.

Every cell of the grid corresponds to a variable. Each cell can take any value between 1 and n. The constraints ensure that two cells on the same row or on the same column are assigned different values. Alternatively, we could also have written the constraints

$$C = \{ \text{allDiff}(x_{i1}, \ldots, x_{in}) \mid 1 \leqslant i \leqslant n \} \cup \{ \text{allDiff}(x_{1j}, \ldots, x_{nj}) \mid 1 \leqslant j \leqslant n \} \quad ,$$

where the allDiff constraint is satisfied when all its arguments are assigned different values.

Solving general CSPs is an NP-complete problem. Constraint Programming (CP) is a complete technique to solve CSPs. Complete means that CP guarantees to find all solutions given enough time.

CP uses a divide-and-conquer strategy. A CSP A is split into smaller CSPs B_i, such that the solution set of A is equal to the union of the solution sets of all B_i. A CSP is smaller if the domain of at least one variable is smaller and the domains of the other variables are smaller or equal.

By recursively splitting the CSPs into smaller CSPs, we obtain a tree. The child nodes are either inconsistent CSPs, or CSPs where the domain of each variable is a

2	1	4	3
1	2	3	4
3	4	1	2
4	3	2	1

Figure 4.1: A Latin square is a simplified Sudoku. Numbers from 1 to n have to be placed in an $n \times n$ grid such that all numbers are different in any row or any column.

singleton. Such a CSP has one trivial solution if the assignment satisfies all the constraints, or no solution otherwise. Solving the original CSP, i.e., finding all solutions, amounts to traversing the whole tree.

To speed up the search, CP exploits the constraints to prune parts of the search tree. At every node, propagators try to detect inconsistent values from the domains of the variables and remove them. A value is inconsistent if it is part of no solution of the CSP. Thus, after a propagation step, the resulting CSP is equivalent, i.e., it has the same solution set, but the domains are smaller than or equal to the original domains. Propagators are called until a fix-point is reached. If a domain becomes empty as a result of propagation, the CSP has no solution and the corresponding node can be pruned.

Detecting inconsistent values may be a hard problem. A trade-off has to be found between the time spent propagating and the time spent searching.

The following sections present the main aspects of a CP solver, i.e., how variables and their domains are represented, how propagators remove inconsistent values, and how the search tree is constructed and traversed.

4.2 Variables and Domains

Variables in CSPs can be of many types: integers, floating-point numbers, sets, graphs, etc. Each variable has an associated domain, i.e., a representation of the set of values that can be assigned to the variable. The representation may be exact or approximate, e.g., by remembering only a lower and upper bound.

Typical implementations of domains include [SC06]:

- Bitsets: the presence of each value in the domain is represented by a bit.
- Bounds: only the lowest and highest values of the domain are remembered. Values in between the bounds are assumed to be present in the domain. With the bounds implementation, we cannot represent holes in the domain.
- Range sequences: an extension of the bounds implementation, allowing holes in the domain. Consecutive values are grouped in a range. The domain is represented by a set of such ranges.

Each node of the search tree is a CSP and should have its own domains. Copying the whole domains for each node is the easiest way to achieve this. However, the search only looks at one node at a time. Thus, only one copy of the domains could be present in memory, along with enough information to restore the domains to any previous node. Such technique is called *trailing*.

A generic trail contains all operations that were performed on the domain, e.g., removing a value. When restoring the domain, the inverse operations are applied. Some domain implementations allow for more efficient specialized trails. For example, the trail of a bitset consists of copies of the bytes that have changed. Hence, multiple operations can be compressed.

4.3 Propagators

The constraints define which assignments of values to the variables are solutions of the CSP. A constraint is a relation between two or more variables. An assignment is a solution of the CSP if it satisfies all the constraints (see definitions 4.2 and 4.3). Some values, called *inconsistent* values, can never be part of a solution and can be removed early on. For example, if x is assigned to 3 and we have the constraint $x \neq y$, we can remove the value 3 from the domain of y.

Definition 4.4. A couple (x, v) is *inconsistent* with respect to constraint c if the CSP $(X, D, \{ c \})$ has no solution μ such that $\mu(x) = v$.

The process of removing inconsistent values from the domains of the variables is called *propagation.* At every node of the search tree, propagation is performed for each constraint individually and is iterated until no values are removed anymore and a fix-point has been reached.

A *propagator* is an algorithm that performs the propagation for a constraint. The rest of this section presents the properties of propagators and how the propagators are called.

4.3.1 Properties of Propagators

Finding all inconsistent values for a constraint can be an NP-hard problem in itself. Propagators can achieve different levels of consistency, depending on how much values they prune [Bes06]. A consistency level is a property of the domains after the propagator has run.

- Checking: no pruning. The propagator only checks if the constraint is satisfied once all variables are assigned.

$$\forall x_i \in \text{vars}(c), D(x_i) = \{ v_i \} \quad \Rightarrow \quad c(v_1, \dots, v_k)$$

- Forward checking: as soon as all variables of the constraint but one are assigned, remove inconsistent values from the domain of the unbound variable.

$$\begin{aligned} &\exists x' \in \text{vars}(c), \forall x_i \in \text{vars}(c) \setminus \{ x' \}, D(x_i) = \{ v_i \} \\ &\quad \Rightarrow \forall v' \in D(x'), c(v_1, \dots, v', \dots, v_k) \end{aligned}$$

- Bound consistency: considering a bound approximation of the domains of the involved variables, ensure the lower and upper bounds are consistent.

$$\begin{aligned} &\forall x' \in \text{vars}(c), \forall v' \in \{ \min(D(x')), \max(D(x')) \}, \\ &\forall x_i \in \text{vars}(c) \setminus \{ x' \}, \exists v_i, \min(D(x_i)) \leqslant v_i \leqslant \max(D(x_i)), \\ &\quad c(v_1, \dots, v', \dots, v_k) \end{aligned}$$

- Domain consistency: ensure every value of the domains of the involved variables appear in at least one solution of the constraint.

$$\begin{aligned}&\forall x' \in \text{vars}(c), \forall v' \in D(x'),\\ &\forall x_i \in \text{vars}(c) \setminus \{ x' \}, \exists v_i \in D(x_i),\\ &\quad c(v_1, \ldots, v', \ldots, v_k)\end{aligned}$$

Note that if all constraints of a CSP are domain consistent, it does not mean that there exists a solution for the CSP. Domain consistent propagators always perform at least the same amount of pruning than the other consistency levels (see fig. 4.2). Forward checking and bound consistency are not comparable (see fig. 4.3).

Two desired properties of propagators are *monotonicity* and *idempotency*. Propagators are monotonic if the order in which they are called does not affect the performed pruning. A propagator is idempotent if it will not perform any more pruning if called immediately after itself.

Like mathematical relations, constraints may be described in *intension* or in *extension*. When described in intension, a specialized propagator is needed to implement the semantics of the constraint. When described in extension, a generic propagator can be used [CY10; Lec11]. Such constraints are called *table* constraints, as the whole table of valid tuples is known in advance.

At some point during the search, a constraint may become *entailed*. An entailed constraint is satisfied by any combination of values from the domains of the variables. For example, if all variables on which the constraint is stated, are assigned, and the constraint is satisfied, then the constraint is entailed.

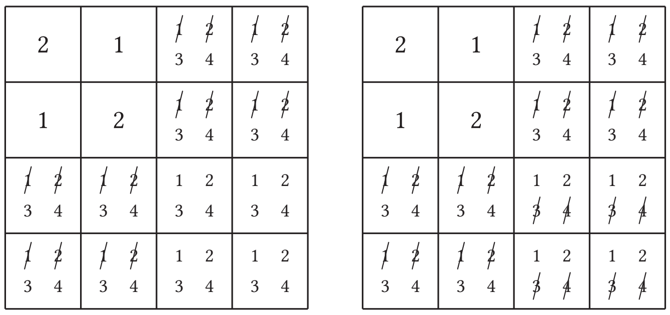
4.3.2 How Propagators are Called

The general propagation algorithm is responsible for calling all propagators until a fix-point is reached. The general propagation algorithm can be constraint-based or value-based.

Constraint-based propagation maintains a queue of propagators, while value-based propagation maintains a queue of propagator-variable-value tuples. In order to prioritize some propagators, multiple queues can be used. As an optimization, entailed constraints may be ignored.

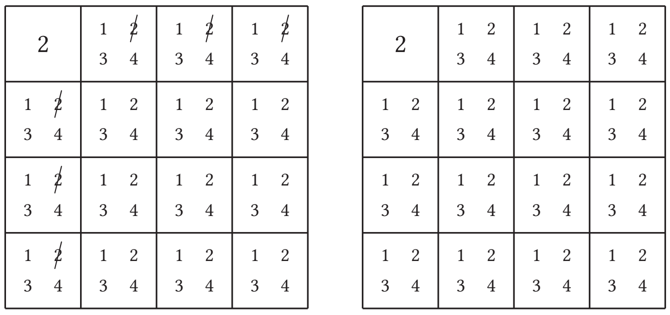
Constraint-based propagation

The constraint-based propagation algorithm maintains a queue of propagators to be called. On initialization, propagators register themselves to events of variables, e.g., when a value is assigned to the variable. When an event occurs, the propagator is added to the queue. Propagators are removed from the queue and called until the queue is empty.



(a) Forward checking (b) Domain consistency

Figure 4.2: Domain consistent propagators can perform more pruning than forward checking propagators. For the domain consistency example, the global allDiff constraint is used on whole rows and whole columns.



(a) Forward checking (b) Bound consistency

Figure 4.3: When the upper left corner is assigned to the value 2, the forward-checking propagator removes that value from the domains of the variables on the same row and on the same column. As the lower bound (1) and upper bound (4) are still consistent, the bound consistent propagator cannot perform any pruning.

The events which the propagators can register to, vary by variable type and domain implementation. For integer variables, standard events are

- bind: when a value has been assigned to the variable, i.e., the domain has become a singleton;
- remove: when a value has been removed from the domain;
- updateBound: when the lower or upper bound has been changed.

While solving a CSP, a propagator can be in one of the following three states:

- propagating, when the propagator is currently being executed (only during the propagation phase),
- queued, when the propagator is in the propagation queue, and
- unqueued, otherwise.

When an event occurs to which a propagator is registered, the propagator will be added to the queue if it is currently unqueued. If the propagator is currently propagating, it will also be added to the queue if it is not idempotent. If the propagator is idempotent, calling it again would perform no more pruning and would only waste time. At last, if the propagator is already queued, nothing is done.

In the worst case, all propagators are queued. Considering only one propagator per constraint, the size of the queue is thus $O(m)$, with m the number of constraints.

Value-based propagation

A value-based propagation algorithm maintains a queue of propagator-variable-value tuples. On initialization, propagators register themselves to variables. When a value v is removed from a variable x, the tuple (p, x, v) is added to the queue for each registered propagator p.

The general propagation algorithm removes a tuple (p, x, v) from the queue and calls the propagator p with arguments x and v. Thus, the propagator only handles the removal of value v from $D(x)$. Such operation can sometimes be done efficiently without traversing the whole domains of the variables. For example, the propagator of constraint $x = y$ removes the value v from the domain of the other variable in constant time. Tuples are removed and propagators are called until the queue is empty.

Tuples are added to the queue unless they are already queued. Because propagators only handle the removal of one value, there is no concept of idempotency unlike for constraint-based propagation.

The queue in value-based propagation is much bigger than the queue in constraint-based propagation. The space complexity is $O(mnd)$, with m the number of constraints, n the number of variables, and d the size of the largest domain.

4.4 Search

The propagation phase, explained in the previous section, allows to reduce the domains of a CSP. Once the fix-point is reached, one has to search for solutions in the remaining solution space. In the search phase, CP divides the solution space by splitting the CSP into smaller CSPs. Propagation is applied on each smaller CSP, which are then split again. By alternating between propagation and search phases, we build a tree of CSPs. The leaf nodes are either solutions, i.e., assignments satisfying all constraints, or a failure. A failure occurs when a propagator detects that a constraint cannot be satisfied.

The search tree can be explored using any standard tree enumeration algorithm, such as breath-first search (BFS) or depth-first search (DFS). To avoid high space complexity, DFS is most often used. Because we have a finite number of variables, the depth of the search tree is also finite. Thus, the DFS algorithm cannot be stuck in an infinite branch. The search is complete.

Along one branch of the depth-first traversal, propagation and search decision operations are applied successively on the domains resulting from the previous operation. Thus, the same domain structures can be used for each node along the branch. When the search backtracks, the structures can be restored using the trailing information.

How the search tree is constructed, is defined by a search heuristic. At each node, the search heuristic selects a variable and splits its domain to generate the child nodes. The search heuristic is thus a combination of a variable selection heuristic and a value selection heuristic. Note that the search tree is never constructed entirely in memory, but rather on-demand while traversing the tree.

Variable selection heuristics follow the first-fail principle. Variables leading to failures quicker should be selected first. The earlier a failure appears in the search tree, the larger the pruned search space and the more efficient the search will be. Some standard variable selection heuristics following the first-fail principle are [Lec09]:

- dom: select the variable with the smallest domain;
- deg: select the variable with the highest degree, i.e., the variable involved in the most constraints;
- ddeg: select the variable with the highest dynamic degree, i.e., the variable involved in the most constraints that are not entailed;
- dom/deg: select the variable with the smallest domain size over degree ratio;
- dom/ddeg: select the variable with the smallest domain size over dynamic degree ratio.

All the above heuristics, except deg, are dynamic so that the ordering of the variables selected by the heuristic depends on the state of the search. The deg heuristic is static as the degree of a variable does not change during the search, except when adding new constraints during the search.

Search heuristics can also be adaptive. An adaptive heuristic learns which variables are difficult during the search. It then changes its behavior to select those variables as early as possible. For example, the wdeg heuristic maintains a weight for each constraint. When a constraint causes the search to fail, its weight is increased. The weight of a variable is defined as the sum of the weights of the constraints in which it is involved. Over time, difficult variables will receive a larger weight.

The value selection heuristic splits the domain of the chosen variable into smaller domains. When every resulting domain is a singleton, the heuristic is called a labeling heuristic. Labeling heuristics are common for finite domains. An alternative is to split the domain in two even parts.

Value selection heuristics are less important than variable selection heuristics when searching for all solutions. Indeed, the order in which the branches will be explored is not important as they will all be explored. This does not hold however when using an adaptive variable selection heuristic. In such cases, or when searching for one solution, the value selection heuristic should select the value that has the greatest probability of participating in a solution.

Finally, one can add constraints during the search. A useful application is the *branch-and-bound* technique, where we look for the solution minimizing some objective function f. As soon as we have found a solution μ, we can post an additional constraint $f(X) < f(\mu)$ in each unexplored node. Next solutions are then guaranteed to have a smaller f-value. The additional constraint can be exploited during the propagation phase to further prune the search space.

4.5 The Comet System

Comet [Dyn10] is a constraint-based optimization system supporting constraint programming, local search, linear programming and mixed integer programming. Optimization problems are encoded in the system with the Comet language. The language is object-oriented and resembles C++ and Java. One notable feature is the ability to write non-deterministic programs to describe the search tree. Because of this, we will use Comet to describe our CP model in chapter 5. This section will briefly present the syntax and core features of Comet with a focus on CP.

A Comet CP program consists of two parts. The first part allows the user to state the problem by means of constraints in a declarative way. The second part is a non-deterministic program that constructs the search tree. Listing 4.1 shows an example solving the Latin square problem of example 4.1. The `solve<cp>{...} using {...}` construct partitions the program in the two parts. The `solve` keyword searches for the first solution. To explore the whole search tree for all solutions, one can use the `solveall` keyword instead.

Constraints are posted with the `post` method. As shown in listing 4.1, constraints may be posted in the declarative part at the root node, or during the search. Con-

```
1  Solver<CP> cp();
2  int N = 4;
3  var<CP>{int} X[1..N,1..N](cp, 1..N);
4  solve<cp> {
5      forall(i in 1..N, j in 1..N, k in (j+1)..N) {
6          cp.post(X[i,j] != X[i,k]);
7          cp.post(X[i,j] != X[k,j]);
8      }
9  } using {
10     forall(i in 1..N, j in 1..N) {
11         tryall<cp>(v in 1..N : X[i,j].memberOf(v))
12             cp.post(X[i,j] == v);
13     }
14     cout << X << endl;
15 }
```

Listing 4.1: A Comet CP program consists of a declarative part where the user posts the constraints (lines 5–8), and a non-deterministic program defining the search tree (lines 10–13). This example solves the Latin square problem formulated in example 4.1.

straints that are posted during the search are removed when backtracking to an ancestor node.

During the search, a binary choice point is introduced with the following structure.

```
try<cp> {
    // left branch
}|{
    // right branch
}
```

In a depth-first search, Comet will first execute the left branch and continue the execution after the `try` block. When backtracking, CP variables and local variables are restored to their state before the `try` block. The right branch is then executed, followed by the code after the `try` block.

A backtrack occurs when a domain becomes empty due to constraint propagation, i.e., the branch has failed. One can also force a branch to fail by calling the `cp.fail()` method. When using the `solveall` structure, the search also backtracks when reaching the end of the `using` block in order to search for other solutions. The whole search can be interrupted with the `cp.exit()` method.

When backtracking, both CP variables (e.g., `X`) and local variables (e.g., `i` and `j`) are restored. However, referenced objects are not restored. Hence, one can use objects to transfer information, such as the number of solutions found so far, between branches. To this end, primitive types (e.g., `int` and `bool`) also exist in object form (e.g., `Integer` and `Boolean`).

The `tryall` structure of lines 11–12 in listing 4.1 is similar to the `try` structure, but introduces a variable number of child nodes. The algorithm described in lines 10–13 of the example is very basic and could be abbreviated by `label(X)`. The built-in `label` function asks Comet to label all the variables, i.e., to try all possible assignments for those variables.

The `Solution` class creates a snapshot of the current solution. That solution can then be restored with the `restore` method.

4.6 Graph Matching with CP

SPARQL queries may be viewed as special kinds of graph matching problems. Graph matching problems consist in finding a matching function between the nodes of two graphs. Let us consider two graphs $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$, with N_1 and N_2 the sets of nodes, and $E_1 \subseteq N_1^2$ and $E_2 \subseteq N_2^2$ the sets of edges. A matching is a function $\mu : N_1 \to N_2$ satisfying some conditions. Common matching problems include:

- *graph homomorphism*, when μ is a total function preserving the edges of G_1, i.e., $\forall (u,v) \in E_1, (\mu(u), \mu(v)) \in E_2$;
- *subgraph isomorphism*, when μ is also injective so that each node of G_1 is matched to at most one node of G_2, i.e., $\forall (u,v) \in N_1^2, u \neq v \Rightarrow \mu(u) \neq \mu(v)$;
- *graph isomorphism*, when μ is a total bijective function and both μ and μ^{-1} are homomorphisms.

The graph homomorphism and subgraph isomorphism problems are known to be NP-complete. Is is unknown if the graph isomorphism problem is NP-complete or in P.

Graph matching problems can easily be modeled by means of CSPs [CDS09]. Each node u of G_1 is represented by a variable x_u. The domain of every variable is the set of nodes of G_2. The constraints encode the specific matching problem. For example, the following CSP models the graph homomorphism problem, which is closely related to the evaluation of basic graph patterns in SPARQL queries.

- $X = \{ x_u \mid u \in N_1 \}$,
- $D(x_u) = N_2 \quad \forall x_u \in X$,
- $C = \{ (x_u, x_v) \in E_2 \mid (u,v) \in E_1 \}$.

Larrosa and Valiente [LV02] have proposed a domain-consistent propagator for the edge preservation constraint. For the subgraph isomorphism problem, more pruning can be obtained by exploiting the fact that a different value must be assigned to every variable [ZDS10; Sol10]. However, the propagators maintain auxiliary data structures that need to be trailed. As will be explained in chapter 6, such structures are impracticable for solving SPARQL queries on large graphs.

Part II

Contributions

Chapter 5

CP Modeling of SPARQL Queries

This chapter covers the reformulation of SPARQL queries by means of Constraint Satisfaction Problems (CSPs). The reformulation gives an alternative denotational semantics of SPARQL. It is then turned into an operational semantics to solve SPARQL queries with CP solvers.

5.1 Denotational CSP Formulation

This section gives a translation of SPARQL semantics for graph pattern evaluation by means of CSPs. By doing so, we transform a declarative semantics (SPARQL) into another declarative semantics (CSP). Hence, we use the term *reformulation* instead of *model*.

To make the section easier to read, we build up the reformulation starting from the straightforward case of basic graph patterns (section 5.1.1). Then, we add simple compositions of basic graph patterns (section 5.1.2). Finally, section 5.1.3 shows the complete semantics for the general case, and proves the equivalence with the SPARQL semantics described in chapter 2.

5.1.1 Basic Graph Patterns and Filters

The translation of a basic graph pattern to a CSP is straightforward. Each variable of the BGP is mapped to a CSP variable. Each triple pattern is a constraint.

Definition 5.1. Let P be a basic graph pattern, i.e., a set of triple patterns, and G an RDF graph. The CSP (X, D, C) associated with (P, G) is defined as follows.

- $X = \text{vars}(P)$,
- $\forall x \in X, D(x) = \mathbb{T}_G$, where $\mathbb{T}_G$ is the set of all RDF terms appearing in G,
- $C = \left\{ \text{Member}\left((s,p,o), G\right) \,\middle|\, (s,p,o) \in P \right\}$, where $\text{Member}(x, S)$ is the set membership constraint which is satisfied if $x \in S$.

Theorem 5.1. *Let P be a basic graph pattern, and G an RDF graph. The set $[\![P]\!]_G$ is the set of solutions of the CSP associated with (P, G).*

Proof. According to definition 2.13, $[\![P]\!]_G \triangleq \{ \mu \mid \mathrm{dom}(\mu) = \mathrm{vars}(P) \wedge \mu[P] \subseteq G \}$. Let μ be a solution mapping such that $\mathrm{dom}(\mu) = \mathrm{vars}(P)$ (but not necessarily $\mu \in [\![P]\!]_G$). Let f_μ be an extension of μ such that

$$\forall x \in \mathrm{dom}(\mu) \cup \mathbb{T}, f_\mu(x) = \begin{cases} \mu(x) & \text{if } x \in \mathrm{dom}(\mu) \\ x & \text{if } x \in \mathbb{T}. \end{cases}$$

We can rewrite the second condition of the definition.

$$\begin{aligned} \mu[P] \subseteq G &\Leftrightarrow \forall (s,p,o) \in \mu[P], (s,p,o) \in G \\ &\Leftrightarrow \forall (s,p,o) \in P, (f_\mu(s), f_\mu(p), f_\mu(o)) \in G \\ &\Leftrightarrow \forall (s,p,o) \in P, \mathrm{Member}\left((s,p,o), G\right) \text{ is satisfied by solution } \mu. \end{aligned}$$

By definition, variables appearing in a Member constraint cannot be assigned a value that does not occur in G. Thus, the domains of the variables can be restricted to $\mathbb{T}_G$. Hence, the set $[\![P]\!]_G$ is equivalent to the set of solutions of the CSP associated with (P, G). □

In the semantics of definition 2.13, the expression in a constrained graph pattern is checked for every solution of the sub-pattern in a post-processing step. When the sub-pattern is a basic graph pattern, such post-processing is equivalent to adding the expression to the set of constraints of the associated CSP. By doing so, the expression can be used to reduce the search space of the CSP.

Definition 5.2. Let P be a basic graph pattern, E an expression, and G an RDF graph. The CSP (X, D, C) associated with $(P \text{ FILTER } E, G)$ is defined as follows.

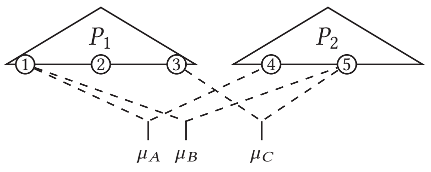
- $X = \mathrm{vars}(P)$,
- $\forall x \in X, D(x) = \mathbb{T}_G$, where $\mathbb{T}_G$ is the set of all RDF terms appearing in G,
- $C = \left\{ \mathrm{Member}\left((s,p,o), G\right) \,\middle|\, (s,p,o) \in P \right\} \cup \{ \mathrm{IsTrue}(E) \}$, where Member is the set membership constraint, and IsTrue is a constraint ensuring the effective Boolean value of an expression is true.

Theorem 5.2. *Let P be a basic graph pattern, E an expression and G an RDF graph. The set $[\![P \text{ FILTER } E]\!]_G$ is the set of solutions of the CSP associated with $(P \text{ FILTER } E, G)$.*

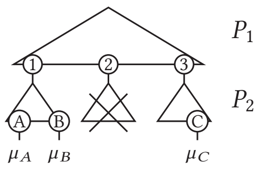
Proof. Trivial from definition 2.13 and theorem 5.1. □

5.1.2 Simple Compound Patterns

Contrarily to classical CSPs, a solution of a compound graph pattern does not have to cover all the variables appearing in the pattern. For example, if a variable x appears



(a) Without replacement semantics



(b) With replacement semantics

Figure 5.1: Evaluating a compound pattern can be done by solving smaller CSPs for every solution of the first sub-pattern. Triangles represent the search trees of the CSP associated with the basic graph patterns.

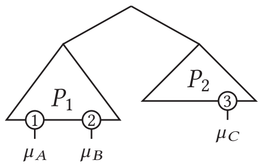
only in an optional part that is not matched in a solution μ, then $x \notin \text{dom}(\mu)$. Such variables are said to be unbound. Hence, compound patterns cannot be directly translated to CSPs. This section explains how to handle such patterns efficiently in the case where the sub-patterns are basic graph patterns. The next section will show the full reformulation for more complex patterns.

To solve compound graph patterns, we could solve all the basic graph patterns separately with CSPs. Then, we can merge the solution sets together following definition 2.13, as illustrated in fig. 5.1a. However, such procedure is inefficient as it keeps all the solution sets in memory, and merging two sets can be costly if the sets are not ordered.

A better way to handle a compound pattern with two sub-patterns P_1 and P_2 is to solve the sub-pattern P_1 first. For every solution μ_1, we solve the sub-pattern $\mu_1[P_2]$, obtained by replacing the variables of μ_1 in P_2 by their values (see fig. 5.1b). The CSPs for $\mu_1[P_2]$ will be smaller and thus usually more efficient to solve.

Disjunctions are introduced by the UNION operator. The solution set of the union of two patterns is the union of the solution sets of both patterns. Basically, the solutions of the two patterns are computed separately.

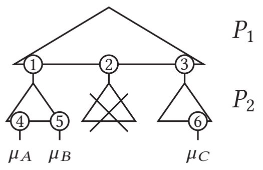
$$[\![P_1 \text{ UNION } P_2]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$$



The figure on the right depicts an example. A triangle represents the search tree of the CSP associated to a basic graph pattern. Circles at the bottom of a triangle are the solutions of the CSP. Circles 1 and 2 represent $[\![P_1]\!]_G$. Circle 3 is the only element in $[\![P_2]\!]_G$.

Two patterns can be concatenated with the AND operator. The solution set of a concatenation is the cartesian product of the solution sets of both patterns. Such cartesian product is obtained by merging every pair of solutions assigning the same values to the common variables. Note that the operator is commutative, i.e., P_1 AND P_2 is equivalent to P_2 AND P_1. The set of solutions is defined as follows.

$$\llbracket P_1 \text{ AND } P_2 \rrbracket_G = \\ \{ \mu_1 \cup \mu_2 \mid \mu_1 \in \llbracket P_1 \rrbracket_G, \mu_2 \in \llbracket \mu_1[P_2] \rrbracket_G \} \ .$$

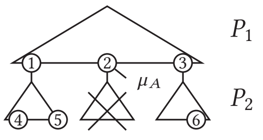


In the example, circles 1, 2 and 3 represent $\llbracket P_1 \rrbracket_G$. Solution 1 is extended into the solutions 4 and 5 in the search tree of $\llbracket \mu_1(P_2) \rrbracket_G$. Solutions 4, 5 and 6 are the solutions of the concatenation.

As P_1 and P_2 are both basic graph patterns, we can compute the concatenation more efficiently by merging both sets of triple patterns. Then, the resulting basic graph pattern can be solved as shown in section 5.1.1. However, such method cannot be extended to the case where P_1 or P_2 are themselved compound patterns. Hence, we have provided the above definition.

The difference of two patterns P_1 and P_2, introduced by the DIFF operator, returns the solutions of P_1 that cannot be extended into a solution of P_2, i.e., each solution μ such that $\mu[P_2]$ is inconsistent. Such inconsistency check makes the search difficult. Indeed, because checking the consistency of a CSP is NP-hard, checking its inconsistency is coNP-hard.

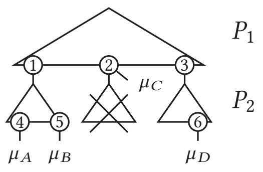
$$\llbracket P_1 \text{ DIFF } P_2 \rrbracket_G = \{ \mu \in \llbracket P_1 \rrbracket_G \mid \llbracket \mu(P_2) \rrbracket_G = \varnothing \}$$



Only circle 2 is a solution of the DIFF pattern because the underlying CSP is inconsistent.

The OPT operator is a combination of the AND and DIFF operators. Intuitively, it solves its left-hand side subpattern P_1 and *tries* to solve its right-hand side subpattern P_2. If a solution of P_1 cannot be extended into a solution of P_1 AND P_2, then that solution of P_1 becomes a solution of the pattern too.

$$\llbracket P_1 \text{ OPT } P_2 \rrbracket_G = \llbracket P_1 \text{ AND } P_2 \rrbracket_G \ \cup \ \llbracket P_1 \text{ DIFF } P_2 \rrbracket_G$$



Compared to the example for the concatenation operator, circle 2 in the figure becomes a solution of the compound pattern.

5.1.3 Complete CSP Formulation

The reformulation described in section 5.1.2 can be directly extended to the general case, where compound patterns may be composed of compound patterns. However, care must be taken with variables that appear in different basic graph patterns. The SPARQL semantics described in section 2.3.2 considers each basic graph pattern separately, and merges the results afterwards. Thus, variables appearing in different basic graph patterns are completely independent until the result sets are joined.

The replacement semantics introduced in section 5.1.2 propagates partial assignments to basic graph patterns before those are evaluated. Hence, variables appearing in different basic graph patterns are not independent anymore. This is not a problem if the solutions have to be compatible anyway, e.g., in an AND pattern. However, in patterns of the form P_1 DIFF P_2, a variable x of P_2 may only be replaced by a value v if x is guaranteed to be part of every solution of P_1. If this is not the case, a solution μ_1 of P_1 that does not assign x might be compatible with a solution μ_2 of P_2, where $\mu_2(x) \neq v$. But that solution does not appear in the evaluation of the substituted P_2. Hence, μ_1 can wrongly become a solution of the pattern. The same condition on the variables of P_2 holds for patterns of the form P_1 OPT P_2. Constrained patterns of the form P' FILTER E have a similar condition on the variables of E.

Example 5.1. Let us consider the RDF graph $\{\,(\texttt{:s}, \texttt{:p}, \texttt{:a}), (\texttt{:t}, \texttt{:p}, \texttt{:b})\,\}$ and three BGPs $P_1 \equiv (x, \texttt{:p}, \texttt{:a})$, $P_2 \equiv (y, \texttt{:p}, \texttt{:a})$, and $P_3 \equiv (x, \texttt{:p}, \texttt{:b})$. We want to evaluate $P \equiv P_1$ AND $(P_2$ DIFF $P_3)$. The sub-pattern P_3 will yield solution $\{\,(x, \texttt{:t})\,\}$, which is compatible with any solution of P_2 as P_2 and P_3 do not share any variables. Hence, the evaluation of P_2 DIFF P_3 is the empty set. The whole pattern P has no solution. Now, if we solve P_1 first and replace every occurrence of x by `:s` in P_2 and P_3, then P_3 has no solutions. The evaluation of P_2 DIFF P_3 returns a solution $\{\,(y, \texttt{:s})\,\}$. The whole pattern then has a solution $\{\,(x, \texttt{:s}), (y, \texttt{:s})\,\}$, which is wrong.

To precisely define which variables may be substituted, we introduce the notion of *unsafe* variables. Intuitively, an unsafe variable is a variable that must not be substituted in order to keep the SPARQL semantics. In example 5.1, variable x is an unsafe variable. The definition of unsafe variables uses another notion, *certain* variables. A certain variable of a pattern is a variable that is always assigned in any solution of the pattern. Note that the notions of unsafe and certain variables are orthogonal. Variable x in example 5.1 is both a certain and an unsafe variable.

Definition 5.3. Let P be a graph pattern, the set of *certain variables* $\mathrm{cvars}(P)$ is recursively defined as follows.

1. If P is a basic graph pattern, $\mathrm{cvars}(P) \triangleq \mathrm{vars}(P)$.
2. If $P \equiv (P_1$ AND $P_2)$, $\mathrm{cvars}(P) \triangleq \mathrm{cvars}(P_1) \cup \mathrm{cvars}(P_2)$.
3. If $P \equiv (P_1$ UNION $P_2)$, $\mathrm{cvars}(P) \triangleq \mathrm{cvars}(P_1) \cap \mathrm{cvars}(P_2)$.
4. If $P \equiv (P_1$ DIFF $P_2)$ or $P \equiv (P_1$ OPT $P_2)$, $\mathrm{cvars}(P) \triangleq \mathrm{cvars}(P_1)$.
5. If $P \equiv (P'$ FILTER $E)$, $\mathrm{cvars}(P) \triangleq \mathrm{cvars}(P')$.

Corollary. *Given a graph pattern P, an RDF graph G and a variable $x \in \mathrm{cvars}(P)$, we have* $\forall \mu \in [\![P]\!]_G, x \in \mathrm{dom}(\mu)$.

Definition 5.4. Let P be a graph pattern, the set of *unsafe variables* $\mathrm{unsafe}(P)$ is recursively defined as follows.

1. If P is a basic graph pattern, $\mathrm{unsafe}(P) \triangleq \emptyset$.
2. If $P \equiv (P_1$ AND $P_2)$ or $P \equiv (P_1$ UNION $P_2)$, $\mathrm{unsafe}(P) \triangleq \mathrm{unsafe}(P_1) \cup \mathrm{unsafe}(P_2)$.
3. If $P \equiv (P_1$ DIFF $P_2)$ or $P \equiv (P_1$ OPT $P_2)$,
   $\mathrm{unsafe}(P) \triangleq \mathrm{unsafe}(P_1) \cup \mathrm{unsafe}(P_2) \cup (\mathrm{vars}(P_2) \setminus \mathrm{cvars}(P_1))$.

4. If $P \equiv (P' \text{ FILTER } E)$, $\text{unsafe}(P) \triangleq \text{unsafe}(P') \cup (\text{vars}(E) \setminus \text{cvars}(P'))$.

The cornerstone of the CSP declarative semantics is lemma 5.5. Basically, the lemma states that the replacement semantics is correct provided we do not substitute unsafe variables. Given a solution mapping μ^*, evaluating a pattern P and keeping only solutions compatible with μ^*, is the same as evaluating P where all variables of μ^* have been replaced by their values.

Before stating lemma 5.5, we first show the distributivity property on the compatibility relation (lemma 5.3) which is needed for the proof. Then, we show a relaxed version (lemma 5.4) when substituting only one variable. Finally, we show the full lemma 5.5.

Lemma 5.3 (Distribution of $\sim$ over $\cup$)**.** *Given three solution mappings μ_1, μ_2 and μ_3,* $(\mu_1 \sim \mu_2) \wedge ((\mu_1 \cup \mu_2) \sim \mu_3) \Leftrightarrow (\mu_1 \sim \mu_2) \wedge (\mu_1 \sim \mu_3) \wedge (\mu_2 \sim \mu_3)$.

Proof. We first prove $(\mu_1 \sim \mu_2) \wedge ((\mu_1 \cup \mu_2) \sim \mu_3) \Rightarrow (\mu_1 \sim \mu_2) \wedge (\mu_1 \sim \mu_3) \wedge (\mu_2 \sim \mu_3)$. By definition, $\mu_1 \sim \mu_3 \Leftrightarrow \forall x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_3), \mu_1(x) = \mu_3(x)$. As $\mu_1 \sim \mu_2$, we have $(\mu_1 \cup \mu_2)(x) = \mu_1(x)$. As $(\mu_1 \cup \mu_2) \sim \mu_3$, we have $(\mu_1 \cup \mu_2)(x) = \mu_3(x)$, and thus $\mu_1(x) = \mu_3(x)$. The same holds for $\mu_2 \sim \mu_3$.

We now prove $(\mu_1 \sim \mu_2) \wedge (\mu_1 \sim \mu_3) \wedge (\mu_2 \sim \mu_3) \Rightarrow (\mu_1 \sim \mu_2) \wedge ((\mu_1 \cup \mu_2) \sim \mu_3)$. As $\mu_1 \sim \mu_2$, the construction $\mu_1 \cup \mu_2$ is valid. By definition, $(\mu_1 \cup \mu_2) \sim \mu_3 \Leftrightarrow \forall x \in \text{dom}(\mu_1 \cup \mu_2) \cap \text{dom}(\mu_3), (\mu_1 \cup \mu_2)(x) = \mu_3(x)$. If $x \in \text{dom}(\mu_1)$, the condition is true because $\mu_1 \sim \mu_3$. If $x \in \text{dom}(\mu_2)$, the condition is true because $\mu_2 \sim \mu_3$. □

Lemma 5.4. *Let P be a graph pattern, G an RDF graph, and $\mu^* = \{ (x^*, v^*) \}$ a solution mapping such that $x^* \notin \text{unsafe}(P)$,*

$$\{ \mu^* \cup \mu \mid \mu \in [\![P]\!]_G \wedge \mu^* \sim \mu \} = \{ \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P]]\!]_G \} \quad .$$

Proof. There are two cases, depending on μ^*. First, when μ^* does not share any variable with P, i.e., $x^* \notin \text{vars}(P)$, then $\mu^*[P] = P$ and μ^* is compatible with any solution of $[\![P]\!]_G$. Thus, lemma 5.4 holds. Second, when $x^* \in \text{vars}(P)$, we build the proof by induction on the number of composition operators in P, i.e., the number of AND, UNION, DIFF, OPT, and FILTER operators appearing in P.
Base case. Let us show that lemma 5.4 holds when P has no composition operator, i.e., P is a basic graph pattern.[1]

$$\begin{aligned}
\text{LHS} &= \{ \mu^* \cup \mu \mid \mu \in [\![P]\!]_G \wedge \mu^* \sim \mu \} \\
&= \{ \mu^* \cup \mu \mid \text{dom}(\mu) = \text{vars}(P) \wedge \mu[P] \subseteq G \wedge \mu^* \sim \mu \} \quad \text{(definition 2.13)} \\
&= \{ \mu^* \cup \mu \mid \text{dom}(\mu) = \text{vars}(P) \wedge \mu[P] \subseteq G \wedge (x^* \notin \text{dom}(\mu) \vee \mu(x^*) = v^*) \} \\
&\qquad \text{(definition 2.8)} \\
&= \{ \mu^* \cup \mu \mid \text{dom}(\mu) = \text{vars}(P) \wedge \mu[P] \subseteq G \wedge \mu(x^*) = v^* \} \quad (x^* \in \text{vars}(P) \Rightarrow x^* \in \text{dom}(\mu))
\end{aligned}$$

[1]Throughout the proofs in this chapter, we will use LHS (resp. RHS) to refer to the left-hand side (resp. right-hand side) of the theorem or lemma.

$$= \{ \mu^* \cup \mu' \mid \mathrm{dom}(\mu') = \mathrm{vars}(P) \setminus \{x^*\} \wedge \mu'[\mu^*[P]] \subseteq G \} \quad \text{(let } \mu' = \mu \setminus \mu^*;\ \mu = \mu' \cup \mu^*)$$
$$= \{ \mu^* \cup \mu' \mid \mathrm{dom}(\mu') = \mathrm{vars}(\mu^*[P]) \wedge \mu'[\mu^*[P]] \subseteq G \}$$
$$= \{ \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P]]\!]_G \} = \mathrm{RHS} \quad \text{(definition 2.13)}$$

Induction hypothesis. Let $k \geqslant 0$. Let us assume that lemma 5.4 holds when P has at most k composition operators.
Inductive step. Let us show that lemma 5.4 holds when P has $k + 1$ composition operators. We have either $P \equiv (P_1 \bullet P_2)$ with $\bullet \in \{ \text{AND}, \text{UNION}, \text{DIFF}, \text{OPT} \}$, or $P \equiv (P_1 \text{ FILTER } E)$. In both cases, P_1 and P_2 have at most k composition operators.
If $P \equiv (P_1 \text{ AND } P_2)$:

$$\mathrm{LHS} = \{ \mu^* \cup \mu \mid \mu \in [\![P]\!]_G \wedge \mu^* \sim \mu \}$$
$$= \left\{ \mu^* \cup \mu \,\middle|\, \mu \in \{ \mu_1 \cup \mu_2 \mid \mu_1 \in [\![P_1]\!]_G \wedge \mu_2 \in [\![P_2]\!]_G \wedge \mu_1 \sim \mu_2 \} \wedge \mu^* \sim \mu \right\}$$
(definition 2.13)
$$= \{ \mu^* \cup \mu_1 \cup \mu_2 \mid \mu_1 \in [\![P_1]\!]_G \wedge \mu_2 \in [\![P_2]\!]_G \wedge \mu_1 \sim \mu_2 \wedge \mu^* \sim (\mu_1 \cup \mu_2) \}$$
$$= \{ \mu^* \cup \mu_1 \cup \mu_2 \mid \mu_1 \in [\![P_1]\!]_G \wedge \mu^* \sim \mu_1 \wedge \mu_2 \in [\![P_2]\!]_G \wedge \mu^* \sim \mu_2 \wedge \mu_1 \sim \mu_2 \}$$
(distribution)
$$= \{ \mu^* \cup \mu_1 \cup \mu_2 \mid \mu_1 \in [\![P_1]\!]_G \wedge \mu^* \sim \mu_1 \wedge \mu_2 \in [\![P_2]\!]_G \wedge \mu^* \sim \mu_2$$
$$\wedge (\mu^* \cup \mu_1) \sim (\mu^* \cup \mu_2) \} \quad \text{(inverse distribution)}$$
$$= \left\{ \mu_1'' \cup \mu_2'' \,\middle|\, \mu_1'' \in \{ \mu^* \cup \mu_1 \mid \mu_1 \in [\![P_1]\!]_G \wedge \mu^* \sim \mu_1 \} \right.$$
$$\left. \wedge \mu_2'' \in \{ \mu^* \cup \mu_2 \mid \mu_2 \in [\![P_2]\!]_G \wedge \mu^* \sim \mu_2 \} \wedge \mu_1'' \sim \mu_2'') \right\}$$
$$= \{ \mu_1'' \cup \mu_2'' \mid \mu_1'' \in \{ \mu^* \cup \mu_1' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \}$$
$$\wedge \mu_2'' \in \{ \mu^* \cup \mu_2' \mid \mu_2' \in [\![\mu^*[P_2]]\!]_G \} \wedge \mu_1'' \sim \mu_2'' \} \quad \text{(induction hypothesis)}$$
$$= \{ \mu^* \cup \mu_1' \cup \mu_2' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \wedge \mu_2' \in [\![\mu^*[P_2]]\!]_G \wedge (\mu^* \cup \mu_1') \sim (\mu^* \cup \mu_2') \}$$
$$= \{ \mu^* \cup \mu_1' \cup \mu_2' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \wedge \mu_2' \in [\![\mu^*[P_2]]\!]_G$$
$$\wedge \mu^* \sim \mu_1' \wedge \mu^* \sim \mu_2' \wedge \mu_1' \sim \mu_2' \} \quad \text{(distribution)}$$
$$= \{ \mu^* \cup \mu_1' \cup \mu_2' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \wedge \mu_2' \in [\![\mu^*[P_2]]\!]_G \wedge \mu_1' \sim \mu_2' \}$$
$(\mathrm{dom}(\mu^*) \cap \mathrm{vars}(\mu^*[P_1]) = \varnothing \Rightarrow \mathrm{dom}(\mu^*) \cap \mathrm{dom}(\mu_1') = \varnothing \Rightarrow \mu^* \sim \mu_1'$; same for μ_2')
$$= \left\{ \mu^* \cup \mu' \,\middle|\, \mu' \in \{ \mu_1' \cup \mu_2' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \wedge \mu_2' \in [\![\mu^*[P_2]]\!]_G \wedge \mu_1' \sim \mu_2' \} \right\}$$
$$= \{ \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P_1] \text{ AND } \mu^*[P_2]]\!]_G \} \quad \text{(definition 2.13)}$$
$$= \{ \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P]]\!]_G \} = \mathrm{RHS}$$

If $P \equiv (P_1 \text{ UNION } P_2)$:

$$\mathrm{LHS} = \{ \mu^* \cup \mu \mid \mu \in [\![P]\!]_G \wedge \mu^* \sim \mu \}$$
$$= \left\{ \mu^* \cup \mu \,\middle|\, \mu \in \{ \mu \mid \mu \in [\![P_1]\!]_G \vee \mu \in [\![P_2]\!]_G \} \wedge \mu^* \sim \mu \right\} \quad \text{(definition 2.13)}$$
$$= \{ \mu^* \cup \mu \mid (\mu \in [\![P_1]\!]_G \vee \mu \in [\![P_2]\!]_G) \wedge \mu^* \sim \mu \}$$
$$= \{ \mu^* \cup \mu \mid (\mu \in [\![P_1]\!]_G \wedge \mu^* \sim \mu) \vee (\mu \in [\![P_2]\!]_G \wedge \mu^* \sim \mu) \} \quad \text{(distribution)}$$

$$= \{\, \mu^* \cup \mu \mid \mu \in [\![P_1]\!]_G \wedge \mu^* \sim \mu \,\} \cup \{\, \mu^* \cup \mu \mid \mu \in [\![P_2]\!]_G \wedge \mu^* \sim \mu \,\}$$
$$= \{\, \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P_1]]\!]_G \,\} \cup \{\, \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P_2]]\!]_G \,\} \quad \text{(induction hypothesis)}$$
$$= \{\, \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P_1]]\!]_G \vee \mu' \in [\![\mu^*[P_2]]\!]_G \,\}$$
$$= \{\, \mu^* \cup \mu' \mid \mu' \in \{\, \mu' \mid \mu' \in [\![\mu^*[P_1]]\!]_G \vee \mu' \in [\![\mu^*[P_2]]\!]_G \,\}\,\}$$
$$= \{\, \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P_1] \text{ UNION } \mu^*[P_2]]\!]_G \,\} \quad \text{(definition 2.13)}$$
$$= \{\, \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P]]\!]_G \,\} = \text{RHS}$$

If $P \equiv (P_1 \text{ DIFF } P_2)$:

$$\text{LHS} = \{\, \mu^* \cup \mu \mid \mu \in [\![P]\!]_G \wedge \mu^* \sim \mu \,\}$$
$$= \left\{ \mu^* \cup \mu \;\middle|\; \mu \in \{\, \mu_1 \mid \mu_1 \in [\![P_1]\!]_G \wedge \neg\exists \mu_2 \in [\![P_2]\!]_G, \mu_1 \sim \mu_2 \,\} \wedge \mu^* \sim \mu \right\}$$
(definition 2.13)
$$= \{\, \mu^* \cup \mu_1 \mid \mu_1 \in [\![P_1]\!]_G \wedge \mu^* \sim \mu_1 \wedge \neg\exists \mu_2 \in [\![P_2]\!]_G, \mu_1 \sim \mu_2 \,\}$$
$$= \{\, \mu^* \cup \mu_1 \mid \mu_1 \in [\![P_1]\!]_G \wedge \mu^* \sim \mu_1 \wedge \neg\exists \mu_2 \in [\![P_2]\!]_G, (\mu^* \cup \mu_1) \sim \mu_2 \,\}$$
($x^* \notin \text{unsafe}(P)$; by definition 5.4, either $x^* \notin \text{vars}(P_2)$ and $\mu^* \sim \mu_2$, or $x^* \in \text{cvars}(P_1)$ and $\mu^* \subseteq \mu_1$)
$$= \left\{ \mu_1'' \;\middle|\; \mu_1'' \in \{\, \mu^* \cup \mu_1 \mid \mu_1 \in [\![P_1]\!]_G \wedge \mu^* \sim \mu_1 \,\} \wedge \neg\exists \mu_2 \in [\![P_2]\!]_G, \mu_1'' \sim \mu_2 \right\}$$
$$= \left\{ \mu_1'' \;\middle|\; \mu_1'' \in \{\, \mu^* \cup \mu_1' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \,\} \wedge \neg\exists \mu_2 \in [\![P_2]\!]_G, \mu_1'' \sim \mu_2 \right\}$$
(induction hypothesis on P_1)
$$= \{\, \mu^* \cup \mu_1' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \wedge \neg\exists \mu_2 \in [\![P_2]\!]_G, (\mu^* \cup \mu_1') \sim \mu_2 \,\}$$
$$= \{\, \mu^* \cup \mu_1' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \wedge \neg\exists \mu_2 \in [\![P_2]\!]_G, \mu^* \sim \mu_2 \wedge \mu_1' \sim \mu_2 \,\} \quad \text{(distribution)}$$
$$= \{\, \mu^* \cup \mu_1' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \wedge \neg\exists \mu_2 \in [\![P_2]\!]_G, \mu^* \sim \mu_2 \wedge \mu_1' \sim (\mu^* \cup \mu_2) \,\}$$
(inverse distribution)
$$= \{\, \mu^* \cup \mu_1' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \wedge \neg\exists \mu^* \cup \mu_2, \mu_2 \in [\![P_2]\!]_G \wedge \mu^* \sim \mu_2 \wedge \mu_1' \sim (\mu^* \cup \mu_2) \,\}$$
$$= \Big\{ \mu^* \cup \mu_1' \;\Big|\; \mu_1' \in [\![\mu^*[P_1]]\!]_G$$
$$\wedge \neg\exists \mu_2'', \mu_2'' \in \{\, \mu^* \cup \mu_2 \mid \mu_2 \in [\![P_2]\!]_G \wedge \mu^* \sim \mu_2 \,\} \wedge \mu_1' \sim \mu_2'' \Big\}$$
$$= \left\{ \mu^* \cup \mu_1' \;\middle|\; \mu_1' \in [\![\mu^*[P_1]]\!]_G \wedge \neg\exists \mu_2'', \mu_2'' \in \{\, \mu^* \cup \mu_2' \mid \mu_2' \in [\![\mu^*[P_2]]\!]_G \,\} \wedge \mu_1' \sim \mu_2'' \right\}$$
(induction hypothesis on P_2)
$$= \{\, \mu^* \cup \mu_1' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \wedge \neg\exists \mu^* \cup \mu_2', \mu_2' \in [\![\mu^*[P_2]]\!]_G \wedge \mu_1' \sim (\mu^* \cup \mu_2') \,\}$$
$$= \{\, \mu^* \cup \mu_1' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \wedge \neg\exists \mu_2' \in [\![\mu^*[P_2]]\!]_G, \mu_1' \sim (\mu^* \cup \mu_2') \,\}$$
$$= \{\, \mu^* \cup \mu_1' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \wedge \neg\exists \mu_2' \in [\![\mu^*[P_2]]\!]_G, \mu_1' \sim \mu^* \wedge \mu_1' \sim \mu_2' \wedge \mu^* \sim \mu_2' \,\}$$
(distribution)
$$= \{\, \mu^* \cup \mu_1' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \wedge \neg\exists \mu_2' \in [\![\mu^*[P_2]]\!]_G, \mu_1' \sim \mu_2' \,\}$$
($x^* \notin \text{dom}(\mu_1') \wedge x^* \notin \text{dom}(\mu_2')$)
$$= \left\{ \mu^* \cup \mu' \;\middle|\; \mu' \in \{\, \mu_1' \mid \mu_1' \in [\![\mu^*[P_1]]\!]_G \wedge \neg\exists \mu_2' \in [\![\mu^*[P_2]]\!]_G, \mu_1' \sim \mu_2' \,\} \right\}$$
$$= \{\, \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P_1] \text{ DIFF } \mu^*[P_2]]\!]_G \,\} \quad \text{(definition 2.13)}$$
$$= \{\, \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P]]\!]_G \,\} = \text{RHS}$$

If $P \equiv (P_1 \text{ OPT } P_2)$, then $[\![P]\!]_G = [\![(P_1 \text{ AND } P_2) \text{ UNION } (P_1 \text{ DIFF } P_2)]\!]_G$. Lemma 5.4 holds because of the cases above.

If $P \equiv (P_1 \text{ FILTER } E)$:

$$\begin{aligned}
\text{LHS} &= \{\, \mu^* \cup \mu \mid \mu \in [\![P]\!]_G \wedge \mu^* \sim \mu \,\} \\
&= \left\{\, \mu^* \cup \mu \;\middle|\; \mu \in \{\, \mu \mid \mu \in [\![P_1]\!]_G \wedge \text{EBV}([\![\mu[E]]\!]) = \mathsf{true} \,\} \wedge \mu^* \sim \mu \,\right\} \\
&\quad \text{(definition 2.13)} \\
&= \{\, \mu^* \cup \mu \mid \mu \in [\![P_1]\!]_G \wedge \text{EBV}([\![\mu[E]]\!]) = \mathsf{true} \wedge \mu^* \sim \mu \,\} \\
&= \{\, \mu^* \cup \mu \mid \mu \in [\![P_1]\!]_G \wedge \text{EBV}([\![\mu[\mu^*[E]]]\!]) = \mathsf{true} \wedge \mu^* \sim \mu \,\} \\
&\quad (x^* \notin \text{unsafe}(P)\text{; by definition 5.4, either } x^* \notin \text{vars}(E) \text{ and } \mu^*[E] = E, \text{ or } x^* \in \text{cvars}(P_1) \text{ and } \mu^* \subseteq \mu) \\
&= \{\, \mu^* \cup \mu \mid \mu \in [\![P_1]\!]_G \wedge \text{EBV}([\![(\mu \cup \mu^*)[E]]\!]) = \mathsf{true} \wedge \mu^* \sim \mu \,\} \\
&= \left\{\, \mu'' \;\middle|\; \mu'' \in \{\, \mu^* \cup \mu \mid \mu \in [\![P_1]\!]_G \wedge \mu^* \sim \mu \,\} \wedge \text{EBV}([\![\mu''[E]]\!]) = \mathsf{true} \,\right\} \\
&= \left\{\, \mu'' \;\middle|\; \mu'' \in \{\, \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P_1]]\!]_G \,\} \wedge \text{EBV}([\![\mu''[E]]\!]) = \mathsf{true} \,\right\} \\
&\quad \text{(induction hypothesis)} \\
&= \{\, \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P_1]]\!]_G \wedge \text{EBV}([\![(\mu' \cup \mu^*)[E]]\!]) = \mathsf{true} \,\} \\
&= \left\{\, \mu^* \cup \mu' \;\middle|\; \mu' \in \{\, \mu' \mid \mu' \in [\![\mu^*[P_1]]\!]_G \wedge \text{EBV}([\![\mu'[\mu^*[E]]]\!]) = \mathsf{true} \,\} \,\right\} \\
&= \{\, \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P_1] \text{ FILTER } \mu^*[E]]\!]_G \,\} \quad \text{(definition 2.13)} \\
&= \{\, \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P]]\!]_G \,\} = \text{RHS}
\end{aligned}$$

□

Lemma 5.5. *Let P be a graph pattern, G an RDF graph, and μ^* a solution mapping such that* $\text{dom}(\mu^*) \cap \text{unsafe}(P) = \emptyset$,

$$\{\, \mu^* \cup \mu \mid \mu \in [\![P]\!]_G \wedge \mu^* \sim \mu \,\} = \{\, \mu^* \cup \mu' \mid \mu' \in [\![\mu^*[P]]\!]_G \,\} \quad .$$

Proof. We build the proof by induction on the cardinality of μ^*.

Base case. Let us show that lemma 5.5 holds when $|\mu^*| = 0$, i.e., $\mu^* = \emptyset$. As the empty mapping is compatible with any mapping and $\mu^*[P] = P$, we have

$$\begin{aligned}
\text{LHS} = \{\, \mu^* \cup \mu \mid \mu \in [\![P]\!]_G \wedge \mu^* \sim \mu \,\} &= \{\, \mu^* \cup \mu \mid \mu \in [\![P]\!]_G \,\} \\
&= \{\, \mu^* \cup \mu \mid \mu \in [\![\mu^*[P]]\!]_G \,\} = \text{RHS} \quad .
\end{aligned}$$

Inductive hypothesis. Let $k \geqslant 0$. Let us assume that lemma 5.5 holds when $|\mu^*| \leqslant k$.

Inductive step. Let us show that lemma 5.5 holds when $|\mu^*| = k + 1$. Let $(x_i, v_i) \in \mu^*$, and $\mu_k = \mu^* \setminus \{\, (x_i, v_i) \,\}$. We have $|\mu_k| = k$. Because $\text{dom}(\mu^*) \cap \text{unsafe}(P) = \emptyset$, $x_i \notin \text{unsafe}(P)$. We denote $\mu_i = \{\, (x_i, v_i) \,\}$. Hence, $\mu^* = \mu_k \cup \mu_i$. We can rewrite the left-hand side of the lemma.

$$\begin{aligned}
\text{LHS} &= \{\, \mu^* \cup \mu \mid \mu \in [\![P]\!]_G \wedge \mu^* \sim \mu \,\} \\
&= \{\, \mu_k \cup \mu_i \cup \mu \mid \mu \in [\![P]\!]_G \wedge (\mu_k \cup \mu_i) \sim \mu \,\} \\
&= \{\, \mu_k \cup \mu_i \cup \mu \mid \mu \in [\![P]\!]_G \wedge \mu_k \sim \mu \wedge \mu_i \sim \mu \,\} \quad \text{(distribution)} \\
&= \{\, \mu_k \cup \mu_i \cup \mu \mid \mu \in [\![P]\!]_G \wedge \mu_k \sim \mu \wedge \mu_i \sim (\mu_k \cup \mu) \,\} \quad \text{(inverse distribution)} \\
&= \left\{\, \mu'' \cup \mu_i \;\middle|\; \mu'' \in \{\, \mu_k \cup \mu \mid \mu \in [\![P]\!]_G \wedge \mu_k \sim \mu \,\} \wedge \mu_i \sim \mu'' \,\right\} \\
&= \{\, \mu'' \cup \mu_i \mid \mu'' \in \{\, \mu_k \cup \mu' \mid \mu' \in [\![\mu_k[P]]\!]_G \,\} \wedge \mu_i \sim \mu'' \,\} \quad \text{(induction hypothesis)} \\
&= \{\, \mu_k \cup \mu_i \cup \mu' \mid \mu' \in [\![\mu_k[P]]\!]_G \wedge \mu_i \sim (\mu_k \cup \mu') \,\} \\
&= \{\, \mu_k \cup \mu_i \cup \mu' \mid \mu' \in [\![\mu_k[P]]\!]_G \wedge \mu_i \sim \mu_k \wedge \mu_i \sim \mu') \,\} \quad \text{(distribution)} \\
&= \{\, \mu_k \cup \mu_i \cup \mu' \mid \mu' \in [\![\mu_k[P]]\!]_G \wedge \mu_i \sim \mu') \,\} \quad (\mu_i \sim \mu_k \text{ by construction})
\end{aligned}$$

Let $P' = \mu_k[P]$. We have $\mu^*[P] = \mu_k[\mu_i[P]] = \mu_i[\mu_k[P]] = \mu_i[P']$. As μ_k and μ_i do not share any variable, the order in which we apply the substitutions does not matter. We have to prove

$$\{\, \mu_k \cup \mu_i \cup \mu' \mid \mu' \in [\![P']\!]_G \wedge \mu_i \sim \mu' \,\} = \{\, \mu_k \cup \mu_i \cup \mu'' \mid \mu'' \in [\![\mu_i[P']]\!]_G \,\} \quad .$$

As the domain of μ_k is disjoint from the domain of μ_i (by construction) and from the domain of any μ' and μ'' (because $\text{dom}(\mu_k) \cap \text{vars}(P') = \varnothing$), the following equation is equivalent.

$$\{\, \mu_i \cup \mu' \mid \mu' \in [\![P']\!]_G \wedge \mu_i \sim \mu' \,\} = \{\, \mu_i \cup \mu'' \mid \mu'' \in [\![\mu_i[P']]\!]_G \,\}$$

This equation is verified by lemma 5.4, because $x_i \notin \text{unsafe}(P')$. □

Thanks to lemma 5.5, we can now define the evaluation of compound patterns using replacement semantics. The definitions are the same as the ones presented in section 5.1.2, except that unsafe variables must not be replaced. Such condition limits the SPARQL queries that our semantics can handle. However, unsafe variables are a counter-intuitive corner case. Real queries are usually free of unsafe substitutions.

Theorem 5.6. *Given a compound pattern* $P \equiv P_1$ AND P_2, *such that* $\text{vars}(P_1) \cap \text{unsafe}(P_2) = \varnothing$, *and an RDF graph G,*

$$[\![P]\!]_G = \{\, \mu_1 \cup \mu_2 \mid \mu_1 \in [\![P_1]\!]_G \wedge \mu_2 \in [\![\mu_1[P_2]]\!]_G \,\} \quad .$$

Proof. By applying lemma 5.5 in definition 2.13, we obtain

$$\begin{aligned}
\text{LHS} = [\![P]\!]_G &= \{\, \mu_1 \cup \mu_2' \mid \mu_1 \in [\![P_1]\!]_G \wedge \mu_2' \in [\![P_2]\!]_G \wedge \mu_1 \sim \mu_2' \,\} \quad \text{(definition 2.13)} \\
&= \left\{\, \mu \;\middle|\; \mu_1 \in [\![P_1]\!]_G \wedge \mu \in \{\, \mu_1 \cup \mu_2' \mid \mu_2' \in [\![P_2]\!]_G \wedge \mu_1 \sim \mu_2' \,\} \,\right\} \\
&= \left\{\, \mu \;\middle|\; \mu_1 \in [\![P_1]\!]_G \wedge \mu \in \{\, \mu_1 \cup \mu_2 \mid \mu_2 \in [\![\mu_1[P_2]]\!]_G \,\} \,\right\} \\
&\qquad \text{(lemma 5.5; } \text{vars}(P_1) \cap \text{unsafe}(P_2) = \varnothing \Rightarrow \text{dom}(\mu_1) \cap \text{unsafe}(P_2) = \varnothing) \\
&= \{\, \mu_1 \cup \mu_2 \mid \mu_1 \in [\![P_1]\!]_G \wedge \mu_2 \in [\![\mu_1[P_2]]\!]_G \,\} = \text{RHS} \quad .
\end{aligned}$$

□

Theorem 5.7. *Given a compound pattern $P \equiv P_1$ UNION P_2, and an RDF graph G,*

$$[\![P]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G \quad .$$

Proof. Trivial from definition 2.13. □

Theorem 5.8. *Given a compound pattern $P \equiv P_1$ DIFF P_2, such that* $\text{vars}(P_1) \cap \text{unsafe}(P_2) = \emptyset$*, and an RDF graph G,*

$$[\![P]\!]_G = \{\, \mu_1 \mid \mu_1 \in [\![P_1]\!]_G \wedge [\![\mu_1[P_2]]\!]_G = \emptyset \,\} \quad .$$

Proof. By applying lemma 5.5 in definition 2.13, we obtain

$$\begin{aligned}
\text{LHS} = [\![P]\!]_G &= \{\, \mu_1 \mid \mu_1 \in [\![P_1]\!]_G \wedge \neg\exists \mu_2 \in [\![P_2]\!]_G , \mu_1 \sim \mu_2 \,\} \quad \text{(definition 2.13)} \\
&= \left\{ \mu_1 \,\middle|\, \mu_1 \in [\![P_1]\!]_G \wedge \neg\exists \mu \in \{\, \mu_1 \cup \mu_2 \mid \mu_2 \in [\![P_2]\!]_G , \mu_1 \sim \mu_2 \,\} \right\} \\
&= \left\{ \mu_1 \,\middle|\, \mu_1 \in [\![P_1]\!]_G \wedge \neg\exists \mu \in \{\, \mu_1 \cup \mu_2' \mid \mu_2' \in [\![\mu_1[P_2]]\!]_G \,\} \right\} \\
&\qquad \text{(lemma 5.5; } \text{vars}(P_1) \cap \text{unsafe}(P_2) = \emptyset \Rightarrow \text{dom}(\mu_1) \cap \text{unsafe}(P_2) = \emptyset\text{)} \\
&= \{\, \mu_1 \mid \mu_1 \in [\![P_1]\!]_G \wedge \neg\exists \mu_2' \in [\![\mu_1[P_2]]\!]_G \,\} \\
&= \{\, \mu_1 \mid \mu_1 \in [\![P_1]\!]_G \wedge [\![\mu_1[P_2]]\!]_G = \emptyset \,\} = \text{RHS} \quad .
\end{aligned}$$

□

An OPT pattern can be rewritten as a combination of AND, UNION, and DIFF operators (see definition 2.13). Hence, the evaluation of the pattern is handled by the theorems above.

Filters on compound patterns cannot be added to the constraint set as in section 5.1.1. Instead, the standard SPARQL semantics have to be applied, i.e., post-processing the filters. For each solution of the pattern, we check if the filter expression is satisfied.

To avoid the post-processing, filters can sometimes be *pushed* down onto the sub-patterns [SML10]. For example, (P_1 UNION P_2) FILTER E can be rewritten as (P_1 FILTER E) UNION (P_2 FILTER E). When P_1 or P_2 is a basic graph pattern, E can be added to the constraint set of the associated CSP.

5.2 Operational CP Modeling

The denotational semantics of SPARQL can be turned into an operational semantics using conventional CP solvers, provided they allow posting constraints during the search.

We will explain the posting of the constraints and the search by means of a non-deterministic program. A non-deterministic program can introduce choice points. At such point, the execution will continue in either the left or the right branch. When the execution arrives at the end of the program or at an explicit failure point, all data structures are restored up to a previous choice point and the execution continues in the other branch. In the pseudo-codes throughout this section, we will use the Comet [Dyn10] notation, introduced in section 4.5.

The proposed operational semantics assumes a depth-first execution. At each choice point, the left branch is explored first. At the end of the program or at a failure point, the search backtracks to the most recently created choice point.

To solve a query instance $((P, M), G)$, we define a global array of CP variables $X = \text{vars}(P)$. The initial domain of each variable $x \in X$ is $D(x) = \mathbb{T}_G$, i.e., all the RDF terms appearing in G. The set of constraints C is initially empty. When no solution modifiers are used, the program solving the query instance is

```
1  solveall<cp> {
2  } using {
3      eval($P$);
4      output($X$);
5  }
```

where line 3 evaluates the graph pattern and line 4 is called for each solution. Note that we do not post any constraint in the declarative part of the program. All constraints are posted during the search.

The next section describes the evaluation of the graph pattern, i.e., the implementation of the `eval` method. Section 5.2.2 explains how solution modifiers are handled.

5.2.1 Graph Pattern Evaluation

The `eval` method is recursively defined for every type of graph pattern. Listing 5.1 shows the `eval` method for basic graph patterns. Filters on a basic graph pattern are posted with the triple patterns. Simple compound graph patterns are shown in listing 5.2. For the concatenation pattern P_1 AND P_2, we solve P_1 first. For every solution μ_1, P_2 is evaluated without restoring the domains of the variables. This effectively computes $\mu_1[P_2]$. The UNION pattern solves the two sub-patterns independently in separate branches.

```
function eval($P$) {
  forall(($s,p,o$) in $P$) {
    cp.post(Member(($s,p,o$),$G$));
  }
  label(vars($P$));
}
```

(a) P is a basic graph pattern

```
function eval($P$ FILTER $E$) {
  forall(($s,p,o$) in $P$) {
    cp.post(Member(($s,p,o$),$G$));
  }
  cp.post(IsTrue($E$));
  label(vars($P$));
}
```

(b) A constrained basic graph pattern

Listing 5.1: A basic graph pattern is a straightforward CP program.

```
function eval($P_1$ AND $P_2$) {
  eval($P_1$);
  eval($P_2$);
}
```

(a) Concatenation

```
function eval($P_1$ UNION $P_2$) {
  try<cp> {
    eval($P_1$);
  }|{
    eval($P_2$);
  }
}
```

(b) Union

Listing 5.2: The AND and UNION operators are easy to implement thanks to the replacement semantics.

The DIFF and OPT patterns, shown in listing 5.3, are similar to the AND pattern. First, P_1 is solved. For every solution μ_1, before evaluating P_2, a choice point is introduced. The left branch computes $[\![\mu_1[P_2]]\!]_G$, hence providing solutions to (P_1 AND P_2). If it succeeds, the right branch is pruned. Otherwise, the right branch is empty and μ_1 is returned as a solution of the OPT pattern. In the case of the DIFF pattern, the left branch is also pruned after the first solution found. Note that the `eval` method here relies on a depth-first search strategy. The left branch must be fully explored before the right branch.

Not every variable is labeled along every branch. The domain of some variables may be untouched when outputting the solution. Such variables are considered unbound and are not part of the solution. Indeed, we always label all variables of a basic graph pattern. Unbound variables do not appear in the basic graph patterns along one branch, due to disjunctions introduced by UNION or differences introduced by DIFF. No constraints are posted on such variables. Their domains are not reduced.

Filters on a compound pattern can only be checked after each solution of the pattern, as shown in listing 5.4. The condition E is not posted as a constraint. Indeed, some variables may be unbound and need to be handled as such.

5.2.2 Handling Solution Modifiers

The complete algorithm for solving a query instance depends on the solution modifiers. In the simpler cases, the solutions are output during the search and forgotten immediately. In more complex cases, e.g., involving sorting, the solutions found need to be stored in memory.

Let X_S, n_O, n_L and $\langle O \rangle$ be respectively the arguments of the PROJECT, OFFSET, LIMIT and ORDER modifiers. In the absence of the corresponding modifier, $X_S = X = \text{vars}(P)$, $n_O = 0$, $n_L = \infty$ and $\langle O \rangle$ is the empty sequence. Listing 5.5 shows the pseudocode corresponding to the two cases. The simple variant handles the PROJECT, OFFSET and LIMIT modifiers as well as the DISTINCT modifier in some cases. The complete variant solves any case at the expense of keeping all the solutions in memory.

```
function eval($P_1$ DIFF $P_2$) {
  eval($P_1$);
  Boolean consistent(false);
  try<cp> {
    eval($P_2$);
    consistent := true;
    cp.fail();
  }|{
    if(consistent)
      cp.fail();
  }
}
```

(a) Difference

```
function eval($P_1$ OPT $P_2$) {
  eval($P_1$);
  Boolean consistent(false);
  try<cp> {
    eval($P_2$);
    consistent := true;
  }|{
    if(consistent)
      cp.fail();
  }
}
```

(b) Optional

Listing 5.3: Compound patterns checking the consistency of a sub-pattern exploit the depth-first search strategy.

```
function eval($P'$ FILTER $E$) {
  eval($P'$);
  if($\text{EBV}([\![\mu[E]]\!]) \neq \text{true}$)
    cp.fail();
}
```

Listing 5.4: Filters on compound patterns need to be post-processed.

```
Integer $n_{\text{sols}}$(0);
**solveall**<cp> {
} **using** {
  eval($P$);
  **if**(DISTINCT) {
    Solution $\mu$(cp);
    cp.postStatic($\bigvee_{x \in X_S} x \neq \mu(x)$);
  }
  **if**($n_{\text{sols}} \geqslant n_O$)
    output($X_S$);
  $n_{\text{sols}}$ := $n_{\text{sols}}$ + 1;
  **if**($n_{\text{sols}} \geqslant n_O + n_L$)
    cp.exit();
}
```

(a) Simple variant: solutions are output during the search.

```
**if**(DISTINCT)
  SortedSet $S(\langle O \rangle$, $X_S$);
**else**
  SortedList $S(\langle O \rangle)$;
**solveall**<cp> {
} **using** {
  eval($P$);
  Solution $\mu$(cp);
  **if**(DISTINCT $\wedge X_S \cap \text{unsafe}(P) = \emptyset$)
    cp.postStatic($\bigvee_{x \in X_S} x \neq \mu(x)$);
  $S$.insert($\mu$);
  **if**($|S| > n_O + n_L$)
    $S$.removeLast();
}
**forall**($\mu$ **in** $S[n_O \mathinner{..} n_O + n_L]$) {
  $\mu$.restore();
  output($X_S$);
}
```

(b) Complete variant: solutions are stored in memory and output at the end.

Listing 5.5: The complete algorithm for solving a query instance depends on the used solution modifiers.

The `SortedSet` and `SortedList` data structures insert solutions in the specified order. The `SortedSet` data structure does not insert solutions that are already present in the set, considering only the specified variables (here, X_S). The `postStatic` method allows to post constraints that will not be removed when backtracking. Instead, those static constraints are reposted after restoring the domains. The `postStatic` method does not exist directly in Comet, but the same effect can be obtained with more complicated structures.

When no selected variables are unsafe, i.e., $X_S \cap \text{unsafe}(P) = \emptyset$, the DISTINCT modifier can be translated to additional static constraints posted after every solution. Such constraints state that any further solution must be different from the solutions so far, considering only the variables in X_S. If a variable in X_S is unsafe, reducing its domain may alter the solutions of a DIFF or FILTER pattern. In such cases, the complete variant without `postStatic` must be used. The `SortedSet` data structure handles the distinctness of the solutions.

When an order is given, the complete variant must be used. Solutions are inserted in a list (or a set if the DISTINCT modifier is specified) at their right places according to the order. If the list grows larger than the offset plus the limit, the extra solutions at the end of the list are discarded.

If the ORDER and LIMIT modifiers are used together, the search can be further optimized with the branch-and-bound technique. After $n_O + n_L$ solutions have been found, we can post an additional static constraint stating that any further solution must be *better* than the worst solution found so far. The meaning of better is defined by the ordering expressions and their directions. Such optimization may only be performed when no variables appearing in the ordering expressions are unsafe. Otherwise, the pruning of the additional constraints may alter the solutions of DIFF or FILTER patterns.

Chapter 6

Castor, a Specialized Lightweight Solver

The operational model described in chapter 5 can be used to solve SPARQL queries with off-the-shelf CP solvers. However, due to the huge domains and triple table, such solvers may not be efficient for the task. This chapter presents Castor, a specialized solver designed to solve SPARQL queries. This solver is experimentally compared with state-of-the-art SPARQL engines and with the Comet CP solver in chapter 7.

The first section presents the data structures that represent the RDF graph. Section 6.2 explains how the domains are implemented in order to cope with their large size. Section 6.3 shows the constraints and their propagators. Section 6.4 describes the search process. Section 6.5 presents the prototype implementation.

6.1 Dataset Representation

The representation of the dataset in Castor is inspired by the RDF-3X engine [NW08]. The main difference is the representation of the values. RDF-3X focuses on triple patterns and does not implement many filters. It considers values to be strings without further interpretation. One of the goals of Castor is to provide efficient filtering. Hence, it needs more information about the values and their interpretation.

The dataset is represented using an on-disk data structure. When Castor loads the dataset, the whole file is memory-mapped. We can thus access any part of the file as if it was in main memory, letting the operating system handle the reading and caching of the file. For performance reasons, it is obviously better to avoid reading the dataset randomly on traditional hard disk drives. The file is partitioned in pages of 16 KB.

Order	Category	Values	Intra-category ordering
1	Blank nodes	$b \in \mathbb{B}$	internal identifier of b
2	IRIs	$i \in \mathbb{I}$	$\mathrm{str}(i)$
3	Simple literals	$a \in \mathbb{L}_{ps}$	$\mathrm{str}(a)$
4	Typed strings	$a \in \mathbb{L}_{ts}$	$\mathrm{str}(a)$
5	Booleans	$a \in \mathbb{L}_{tb}$	$(\mathrm{value}(a), \mathrm{str}(a))$
6	Numbers	$a \in \mathbb{L}_{tn}$	$(\mathrm{value}(a), \mathrm{datatype}(a), \mathrm{str}(a))$
7	Dates	$a \in \mathbb{L}_{td}$	$(\mathrm{value}(a), \mathrm{str}(a))$
8	Other plain literals	$a \in \mathbb{L}_{pl}$	$(\mathrm{lang}(a), \mathrm{str}(a))$
9	Other typed literals	$a \in \mathbb{L}_{to}$	$(\mathrm{datatype}(a), \mathrm{str}(a))$

Table 6.1: Values are partitioned in nine categories, which are shown in ascending order. Inside each category, values are ordered according to the key given in the last column. When a couple of keys is given, the order is lexicographic. Because multiple literals can have the same interpreted value, we break ties by ordering on the uninterpreted lexical form.

6.1.1 Mapping RDF Values to Integer Identifiers

Like in RDF-3X, values are mapped to consecutive integers. The mapping function, however, differs from RDF-3X. Let $\mathrm{id}(v)$ be the identifier mapped to the RDF value v. To efficiently implement a bound consistent propagator for inequality filters, we want $v_1 <_F v_2 \Rightarrow \mathrm{id}(v_1) < \mathrm{id}(v_2)$, where $<_F$ is the partial order operator for comparing SPARQL expressions (see definition 2.11). The SPARQL specification only defines $<_F$ between numerical values, between simple literals, between strings, between Boolean values, and between dates.

To efficiently implement the ORDER solution modifier, we also want $v_1 <_O v_2 \Rightarrow \mathrm{id}(v_1) < \mathrm{id}(v_2)$, where $<_O$ is the partial order defined in the SPARQL specification. This order introduces a precedence between blank nodes, URIs and literals. Literals are ordered with respect to $<_F$, so that $v_1 <_F v_2 \Rightarrow v_1 <_O v_2$.

To map each RDF value to a unique identifier, we introduce a total order $<_T$ that is compatible with both partial orders, i.e.,

$$\forall (v_1, v_2) \in \mathbb{T} \times \mathbb{T}, v_1 <_O v_2 \Rightarrow v_1 <_T v_2 \quad .$$

Values are partitioned into categories as shown in table 6.1. Inside each category, values are ordered according to the rules specific to the category. The total order $<_T$ is defined as the lexicographic order of the inter- and intra-category ordering.

We map the values of a dataset to consecutive integers starting from 1, such that $v_1 <_T v_2 \Leftrightarrow \mathrm{id}(v_1) < \mathrm{id}(v_2)$. Table 6.2 shows the mapping of the terms appearing in our running example to identifiers. Identifiers in Castor are encoded with 32 bits. Hence, a dataset cannot contain more than 4 billion values. If more values are needed, one could enlarge identifiers to 64 bits.

Category	Identifier	Term
1	1	`_:a`
2	2	`http://dbpedia.org/resource/Procrastination`
2	3	`http://dbpedia.org/resource/Research`
2	4	`http://phdcomics.com/#Cecilia`
2	5	`http://phdcomics.com/#Free Food`
2	6	`http://phdcomics.com/#Mike`
2	7	`http://phdcomics.com/#Smith`
2	8	`http://phdcomics.com/#Students`
2	9	`http://phdcomics.com/#Tajel`
2	10	`http://purl.org/dc/terms/created`
2	11	`http://purl.org/dc/terms/subject`
2	12	`http://www.phdcomics.com/blog.php`
2	13	`http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq`
2	14	`http://www.w3.org/1999/02/22-rdf-syntax-ns#_1`
2	15	`http://www.w3.org/1999/02/22-rdf-syntax-ns#_2`
2	16	`http://www.w3.org/1999/02/22-rdf-syntax-ns#type`
2	17	`http://www.w3.org/2001/XMLSchema#dateTime`
2	18	`http://www.w3.org/2001/XMLSchema#decimal`
2	19	`http://www.w3.org/2001/XMLSchema#integer`
2	20	`http://www.w3.org/2001/XMLSchema#string`
2	21	`http://xmlns.com/foaf/0.1/Group`
2	22	`http://xmlns.com/foaf/0.1/age`
2	23	`http://xmlns.com/foaf/0.1/interest`
2	24	`http://xmlns.com/foaf/0.1/knows`
2	25	`http://xmlns.com/foaf/0.1/member`
2	26	`http://xmlns.com/foaf/0.1/name`
2	27	`http://xmlns.com/foaf/0.1/topic`
2	28	`http://xmlns.com/foaf/0.1/weblog`
3	29	("Brian B. Smith", "")
3	30	("Michael Slackenerny", "")
3	31	("Tajel", "")
3	32	("comics", "")
3	33	("procrastination", "")
4	34	("Cecilia", `xsd:string`)
6	35	("26", `xsd:integer`)
6	36	("29", `xsd:integer`)
6	37	("35", `xsd:decimal`)
6	38	("56", `xsd:integer`)
7	39	("2005-07-10T08:20:00", `xsd:dateTime`)

Table 6.2: The RDF terms appearing in the running example of fig. 2.2 are mapped to consecutive integers.

Categories that can be interpreted by SPARQL, i.e., strings, booleans, numbers and dates, are ordered by their interpreted value. Hence, equivalent values that are not identical, are assigned to consecutive identifiers. For example, ("1",`xsd:integer`), ("1",`xsd:float`), and ("1.0",`xsd:float`) will have consecutive identifiers.

6.1.2 Representing an RDF Value

The strings appearing in the dataset, i.e., IRI strings, lexical forms and language tags, are stored separately from values. As for values, each string is mapped to a consecutive integer identifier, starting from 1. The string table is stored on disk along with a B$^+$-tree mapping hash values to the corresponding string identifiers.

A value on disk is a fixed-size data structure with the following fields.

- the 32-bits identifier,
- the category (see table 6.1),
- the numerical subcategory for numbers: integers, floating point numbers or decimal numbers with arbitrary precision,
- the identifier of the datatype (pointing to an IRI value),
- the identifier of the language tag or the lexical form of the datatype (pointing to a string),
- the identifier of the lexical form (pointing to a string),
- a 64-bit integer approximation if the value is a number.

The numerical subcategory indicates how the lexical form must be converted to a number. For typed literals, the lexical form of the datatype gives quick access to the string representation of the datatype IRI. It is redundant with the datatype in the previous field. The integer part of a numeric value is encoded in the last field. Such approximation is used to propagate arithmetic constraints.

The value table is stored on disk along with a B$^+$-tree mapping hash values to the corresponding value identifiers. The hash value of an RDF value is computed with the category, numerical subcategory, language tag or lexical form of the datatype, and lexical form fields.

Values that are not identical, may still be equivalent. For example, the integer ("1",`xsd:integer`) and the floating-point number ("1.0",`xsd:float`) are equivalent, but are different values with different identifiers. The *equivalence class* of a value v is the set of values that are equivalent to v. Equivalent values have neighboring identifiers. Thus, the equivalence class of a value is represented as a simple range of identifiers.

To quickly look up the equivalence class of a value, we store a bitmap on disk with one bit per value. Bit i is 1 if the value with identifier i is the first value in its equivalence class, and 0 otherwise. Hence, finding the range of identifiers that are equivalent to a value, amounts to finding the indexes of the preceding and following bit 1. Such bit-wise operations are handled efficiently by the CPU.

Example 6.1. Consider the numeric values ("0",`xsd:integer`), ("1",`xsd:integer`), ("1.0",`xsd:float`), ("1.00",`xsd:float`), and ("2",`xsd:float`). The associated bitmap is `11001`. The equivalence class for the third value (1.0) is represented by the identifier range $[2,4]$.

6.1.3 Representing Triples

RDF triples are represented by three integers. To provide efficient look-up, triples are stored three times in different lexicographical orders: SPO, POS and OSP, where S stands for subject, P for predicate and O for object. For example, SPO means the triples are ordered first by subject, then by predicate and finally by object.

The sorted triples are written on disk as B^+-trees. One page corresponds to one node of the tree. Leaves are linked together to allow efficient in-order traversal. The triples inside a leaf are compressed using the delta-algorithm described in Neumann and Weikum [NW08].

Because the same leaf pages are often used in the propagators, and decompressing a leaf is costly, Castor keeps a small cache of the most recently used decompressed pages.

Given two triples $t_{min} = (s_{min}, p_{min}, o_{min})$ and $t_{max} = (s_{max}, p_{max}, o_{max})$, the $\textsc{Fetch}(t_{min}, t_{max})$ operation returns the set of triples between t_{min} and t_{max}, according to the lexicographic order of the index specified in table 6.3. When all components are fixed, i.e., $t_{min} = t_{max}$, the set is a singleton if the triple appears in the dataset and empty otherwise. When only one component is not fixed, e.g., $s_{min} = s_{max}$ and $p_{min} = p_{max}$ but $o_{min} < o_{max}$, that component is guaranteed to be within its bounds in the returned triples. However, when more than one component is not fixed, no such property can be guaranteed, due to the in-order traversal of the B^+-tree.

Example 6.2. Consider the graph $G = \{(1,2,5),(1,2,7),(1,3,4),(1,3,8)\}$. Executing the $\textsc{Fetch}((1,2,3),(1,2,6))$ operation will use the SPO index because both subject and predicate are fixed. The single result is the triple $(1,2,5)$. The $\textsc{Fetch}((1,2,7),(1,3,8))$ operation will also use the SPO index because only the subject is fixed. In this case, the result set is $\{(1,2,7),(1,3,4),(1,3,8)\}$. Note the presence of triple $(1,3,4)$ even though object 4 is smaller than the requested lower bound 7.

In addition to the triple indexes, the whole table is also stored uncompressed. The table is used by propagators needing direct access to the whole table, such as the domain-consistent triple propagator (see section 6.3.1).

6.2 Variables and Domains

Each variable in a SPARQL query is a decision variable in Castor. Domains of decision variables are identifiers of all RDF values appearing in the dataset, along with the

$s_{min} = s_{max}$	$p_{min} = p_{max}$	$o_{min} = o_{max}$	Index
Yes	Yes	Yes	POS
Yes	Yes	No	SPO
Yes	No	Yes	OSP
Yes	No	No	SPO
No	Yes	Yes	POS
No	Yes	No	POS
No	No	Yes	OSP
No	No	No	POS

Table 6.3: Given the triples $t_{min} = (s_{min}, p_{min}, o_{min})$ and $t_{max} = (s_{max}, p_{max}, o_{max})$, the index is chosen so that the fixed components come first in the ordering. Three different orderings are sufficient to cover all cases. Note that the index of the first and last row does not matter for correctness. POS has been chosen to maximize the cache usage.

special value 0. As long as 0 is in the domain of a variable, that variable is considered unbound in the SPARQL semantics and is not part of the solution mapping.

When posting filters, auxiliary variables are introduced, as explained in section 6.3.2. The results of predicates are represented by Boolean variables. To conform to the SPARQL semantics, the domain of Boolean variables have three values: true, false and error. Arithmetic constraints are posted on numeric variables whose domains only contain integers. Integers were chosen over floating point numbers to avoid numerical instabilities.

During the search, domains get reduced at each choice point and restored when backtracking. The data structures representing domains should perform such operations efficiently. There are two kinds of representations. The *discrete* representation keeps track of every single value in the domain. The *bound* representation only keeps the lowest and highest value of the domain.

Boolean auxiliary variables use the discrete representation. Numeric auxiliary variables use the bound representation, as it is impossible to store all possible integers in main memory. For decision variables, we propose a dual view, leveraging the strengths of both representations.

6.2.1 Discrete Representation

The discrete representation is based on sparse sets presented by Briggs and Torczon [BT93]. The domain of each variable x is represented by its size $size_x$ and two arrays dom_x and map_x. The $size_x$ first elements of dom_x are in the domain of x, the others have been removed (see fig. 6.1). The map_x array maps values to their position in the dom_x array.

Note that this is not the standard representation of discrete domains in CP. How-

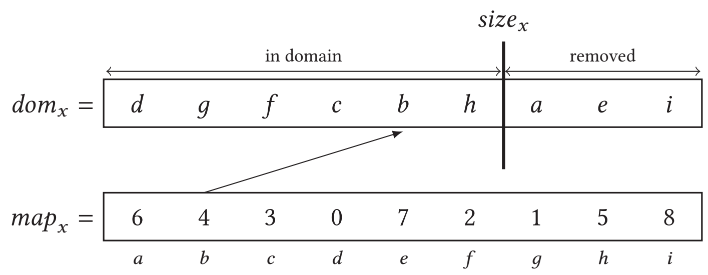


Figure 6.1: Example representation of the domain $\{ b, c, d, f, g, h \}$, such that $size_x = 6$, when the initial domain is $\{ a, \ldots, i \}$. The $size_x$ first values in dom_x belong to the domain; the other values have been removed earlier. The map_x array maps values to their position in dom_x. For example, value b has index 4 in the dom_x array. In such representation, only the size needs to be kept in the trail.

ever, the trail, i.e., the data structure needed to restore the domain to any ancestor node of the search tree, of standard representations is too heavy for our purpose and size of data.

For a variable x, the following invariants hold:

- $D(x) = \{ dom_x[i] \mid 0 \leqslant i < size_x \}$
- $map_x[v] = i \Leftrightarrow dom_x[i] = v$
- The values in $dom_x[size_x \,..\, cap_x - 1]$ are not modified by any operation, where cap_x is the size of the dom_x array, i.e., the size of the initial domain.

Thanks to the last invariant, the domain can be restored in constant time by setting the $size_x$ marker back to its previous position. The trail is thus a stack of the sizes.

To remove a value, we swap it with the last value of the domain (i.e., the value directly to the left of the $size_x$ marker), reduce $size_x$ by one and update the map_x array. Such operation is done in constant time, as shown in algorithm 6.1.

Alternatively, we can restrict the domain to the intersection of itself and a set M. We first move all values of M which belong to the $size_x$ first elements of dom_x, i.e., which are still in the domain, at the beginning of dom_x. Such operation is called Mark in algorithm 6.1. The $mark_x$ counter keeps track of the marked values (see fig. 6.2). We denote the set of marked values by M_x. Once all values are marked, we set $size_x$ to the size of the intersection, i.e., $mark_x$. The whole operation is done in $O(|M|)$, with $|M|$ the size of M. Castor uses the restriction operation in propagators achieving forward-checking consistency.

Operations on the bounds however are inefficient. This major drawback is due to the unsorted dom_x array. Searching for the minimum or maximum value requires the traversal of the whole domain. Increasing the lower bound or decreasing the upper bound involves removing every value between the old and new bound one by one.

```
function Contains($x, v$)                                  ▷ $v \in D(x)$
    Return $map_x[v] < size_x$
end function

procedure Swap($x, i, j$)                 ▷ Swap elements at positions $i$ and $j$ in $dom_x$.
    $dom_x[i], dom_x[j] \leftarrow dom_x[j], dom_x[i]$
    $map_x[dom_x[i]] \leftarrow i$
    $map_x[dom_x[j]] \leftarrow j$
end procedure

procedure Bind($x, v$)                                     ▷ $D(x) \leftarrow D(x) \cap \{ v \}$
    if $map_x[v] \geqslant size_x$ then
        $size_x \leftarrow 0$
    else
        Swap($x, map_x[v], 0$)
        $size_x \leftarrow 1$
    end if
end procedure

procedure Remove($x, v$)                                   ▷ $D(x) \leftarrow D(x) \setminus \{ v \}$
    if $map_x[v] < size_x$ then
        Swap($x, map_x[v], size_x - 1$)
        $size_x \leftarrow size_x - 1$
    end if
end procedure

procedure ClearMarks($x$)                                  ▷ $M_x \leftarrow \varnothing$
    $mark_x \leftarrow 0$
end procedure

procedure Mark($x, v$)                                     ▷ $M_x \leftarrow M_x \cup (D(x) \cap \{ v \})$
    if $map_x[v] < size_x \wedge map_x[v] >= mark_x$ then
        Swap($x, map_x[v], mark_x$)
        $mark_x \leftarrow mark_x + 1$
    end if
end procedure

procedure Restrict($x$)                                    ▷ $D(x) \leftarrow M_x$
    $size_x \leftarrow mark_x$
end procedure
```

Algorithm 6.1: Operations on the discrete representation of variables involve swapping values in the dom_x array. All procedures have $O(1)$ time complexity.

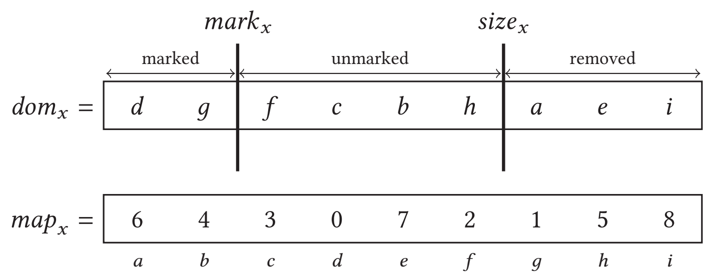


Figure 6.2: Values can be marked in the discrete representation of a domain by moving them to the beginning of the dom_x array and increasing the $mark_x$ marker. To restrict the domain to only the marked values, we set $size_x$ to $mark_x$.

6.2.2 Bound Representation

The domain of every variable x is also represented by its bounds, i.e., its minimum and maximum values. In contrast to the discrete representation, the bound representation is an approximation of the exact domain. We assume all values between the bounds are present in the domain.

In such a representation, we cannot remove a value in the middle of the domain as we cannot represent a hole inside the bounds. However, increasing the lower bound or decreasing the upper bound is done in constant time.

The data structure for this representation being small (only two numbers), the trail contains copies of the whole data structure. Restoring the domains involves restoring both bounds.

6.2.3 Dual View

Propagators achieving forward-checking or domain consistency remove values from the domains. Thus, they require a discrete representation. However, propagators achieving bound consistency only update the bounds of the domains. For them to be efficient, we need a bound representation. Hence, Castor creates two variables x_D and x_B (resp. with discrete and bound representation) for every SPARQL variable x. Constraints are stated using only one of the two variables, depending on which representation is the most efficient for the associated propagator. In particular, arithmetic inequality constraints are stated on x_B whereas triple pattern constraints are stated on x_D.

An additional constraint $x_D = x_B$ ensures the correctness of the dual approach. Achieving domain consistency for this constraint is too costly, as it amounts to perform every operation on the bounds also on the discrete representation. Instead, the propagator in Castor achieves forward-checking consistency, i.e., once one variable is assigned, the same value will be assigned to the other variable.

For practical reasons, the two representations x_D and x_B are implemented as one

```
**procedure** ClearMarks($x$)                                   ▷ $M_x \leftarrow \emptyset$
    $mark_x \leftarrow 0$
    $mark_x^{min}, mark_x^{max} \leftarrow +\infty, -\infty$
**end procedure**

**procedure** Mark($x, v$)                        ▷ $M_x \leftarrow M_x \cup (D(x) \cap \{ v \})$
    **if** $map_x[v] < size_x \wedge map_x[v] >= mark_x \wedge min_x \leqslant v \leqslant max_x$ **then**
        Swap($x, map_x[v], mark_x$)
        $mark_x \leftarrow mark_x + 1$
        $mark_x^{min} \leftarrow \min(mark_x^{min}, v)$
        $mark_x^{max} \leftarrow \max(mark_x^{max}, v)$
    **end if**
**end procedure**

**procedure** Restrict($x, v$)                                  ▷ $D(x) \leftarrow M_x$
    $size_x \leftarrow mark_x$
    $min_x, max_x \leftarrow mark_x^{min}, mark_x^{max}$
**end procedure**
```

Algorithm 6.2: In the dual representation, the Restrict operation is done on both representations at the same time without additional complexity.

object. The propagation of the channeling constraint $x_D = x_B$ is embedded inside the Bind operation. As an optimization, when restricting a domain to its intersection with a set M, we filter out values of M which are outside the bounds and update the bounds of x_B. Such optimization does not change the complexity of the operation, as it has to traverse the whole set M anyway. The new operation is shown in algorithm 6.2.

6.3 Constraints and Propagators

The domains of the variables can be huge. A value-based propagation queue can grow very large. To avoid this problem, Castor uses constraint-based propagation queues. There are three priority levels and a separate queue for each of them. Constraints with higher priority are always propagated before lower-priority constraints. Constraints are given priorities based on the consistency level they achieve:

- Highest priority: domain consistent and bound consistent propagators. Such propagators are very fast. Bound consistent propagators have constant time complexity. Domain consistent propagators behave linearly with respect to the number of removed values.
- Medium priority: forward checking propagators. Such propagators are entailed after they are called. They do not need to be called again later in the branch. However, they have to iterate over all the values of a domain.

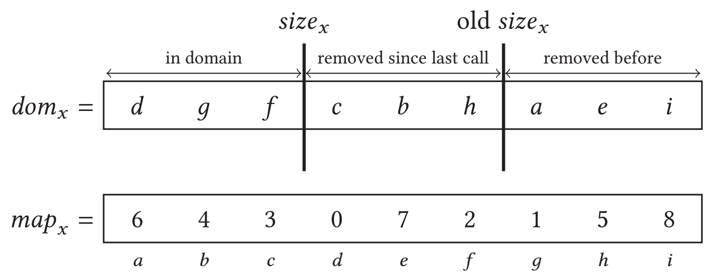


Figure 6.3: If “old $size_x$” was the size of the domain of x on the last propagation of a constraint, the values c, b and h have been removed since then. To perform value-based propagation, the propagator only has to check the values for which c, b or h was a support. Note that the “old $size_x$” marker is a property of the propagator and is different for each propagator, while $size_x$ is a property of the domain of x.

- Lowest priority: non-monotonic propagators. The order in which they are called affect the amount of pruning they perform. The smaller the domains, the better the pruning. Because they also have a high complexity, similar to forward checking propagators, they are called last.

While the propagation queues are constraint-based, it is still possible to perform value-based propagation in a constraint. At the end of the propagation of a constraint, we remember the sizes of the domains. On the next propagation, the values that have been removed from the domains will be between the saved $size_x$ and the new $size_x$, as shown in fig. 6.3. By iterating over the values between the two markers, we effectively perform value-based propagation. When backtracking, the saved sizes must be reset.

All propagators in Castor are considered to be idempotent, i.e., calling the same propagator again immediately after it has finished, does not perform any more pruning. Thus, the currently running propagator is never added back in the propagation queue.

6.3.1 Triple Patterns

The most used constraint when solving a SPARQL query is the triple pattern. A triple pattern involves three variables x_s, x_p and x_o. It is satisfied if the triple (x_s,x_p,x_o) is present in the dataset. Basically, a triple pattern is a table constraint with arity 3 and the table being the whole dataset. The difficulty arises from the size of the domains and of the table. Propagators maintaining supports for every variable-value pair, require too much memory and are too heavy to backtrack. Thus, they are not considered. In this section, we present three propagators for the triple pattern constraint, achieving different levels of consistency.

The forward-checking propagator (FC) is shown in algorithm 6.3. When all variables but one are assigned, the domain of the unassigned variable is restricted to the values for which there exists a support triple in the dataset. The Fetch operation

```
procedure TripleFC(($x_s, x_p, x_o$))
    if there is at most one unassigned variable in $\{ x_s, x_p, x_o \}$ then
        $t_{min} \leftarrow (min_{x_s}, min_{x_p}, min_{x_o})$
        $t_{max} \leftarrow (max_{x_s}, max_{x_p}, max_{x_o})$
        $T \leftarrow$ Fetch($t_{min}, t_{max}$)
        ClearMarks($x_s$)
        ClearMarks($x_p$)
        ClearMarks($x_o$)
        for all $(s, p, o) \in T$ do
            Mark($x_s, s$)
            Mark($x_p, p$)
            Mark($x_o, o$)
        end for
        Restrict($x_s$)
        Restrict($x_p$)
        Restrict($x_o$)
    end if
end procedure
```

Algorithm 6.3: The forward-checking propagator for a triple pattern waits for all but one variable to be assigned.

is called to fetch the support triples (see section 6.1.3). The time complexity of the propagator is $O(t)$ with t the number of triples fetched from the store.

Additional pruning can be obtained by propagating as soon as one variable is assigned. Such a propagator, called FC+, is shown in algorithm 6.4. The propagator achieves forward-checking consistency, like the FC propagator. However, when one variable is bound, it also achieves one-time domain consistency. The domain consistency is not maintained in further propagation calls. This is similar to nFC2 described by Bessière et al. [Bes+99]. As more than one variable may be unassigned, the database Fetch operation may return triples that are outside the given bounds, as explained in section 6.1.3. Thus, we check that every component of the support triple does appear in the domain of the corresponding variable. Consider for example the domains $D(x_s) = \{ 1 \}$, $D(x_p) = \{ 2, 3 \}$, $D(x_o) = \{ 2, 4 \}$ and the support triples $T = \{ (1, 2, 3), (1, 3, 4) \}$. Without the check, the domain of x_p would not be reduced, even though there are no supports in $D(x_o)$ for $x_p = 2$.

Even more pruning can be obtained with the domain-consistent propagator (DC) shown in algorithm 6.5. The propagator is an instance of the STR algorithm [Lec11] for tables of arity 3. The trailable set of support triples S initially contains all the triples of the dataset. The set is implemented as a sparse set, similar to the discrete representation of variable. Hence, only its size has to be trailed. The time complexity of the DC propagator is $O(t)$ with t the number of triples in the support set. As the propagator is called whenever a domain changes, the cost is quite high, as will be

```
procedure TripleFC+(($x_s$,$x_p$,$x_o$))
    if there are at most two unassigned variables in { $x_s$,$x_p$,$x_o$ } then
        $t_{min} \leftarrow (min_{x_s}, min_{x_p}, min_{x_o})$
        $t_{max} \leftarrow (max_{x_s}, max_{x_p}, max_{x_o})$
        $T \leftarrow$ Fetch($t_{min}$,$t_{max}$)
        ClearMarks($x_s$)
        ClearMarks($x_p$)
        ClearMarks($x_o$)
        for all $(s,p,o) \in T$ do
            if $s \in D(x_s) \land p \in D(x_p) \land o \in D(x_o)$ then
                Mark($x_s$,$s$)
                Mark($x_p$,$p$)
                Mark($x_o$,$o$)
            end if
        end for
        Restrict($x_s$)
        Restrict($x_p$)
        Restrict($x_o$)
    end if
end procedure
```

Algorithm 6.4: The extended forward-checking propagator for a triple pattern additionally achieve domain consistency once when at least one variable is assigned. It does not maintain the domain consistency property.

```
procedure TripleDC($(x_s, x_p, x_o)$)
    ClearMarks($x_s$)
    ClearMarks($x_p$)
    ClearMarks($x_o$)
    for all $(s,p,o) \in S$ do
        if $s \in D(x_s) \land p \in D(x_p) \land o \in D(x_o)$ then
            Mark($x_s, s$)
            Mark($x_p, p$)
            Mark($x_o, o$)
            if $\forall x \in \{ x_s, x_p, x_o \}, mark(x) = size(x)$ then
                Return
            end if
        else
            $S \leftarrow S \setminus \{ (s,p,o) \}$
        end if
    end for
    Restrict($x_s$)
    Restrict($x_p$)
    Restrict($x_o$)
end procedure
```

Algorithm 6.5: The domain-consistent propagator is an instance of the STR algorithm [Lec11]. It maintains a support set S using a sparse set data structure similar to the discrete domains.

shown in section 7.4.3. Hence, the DC propagator is not used in Castor.

6.3.2 Filter Expressions

The second type of constraints in Castor are expressions used as filters. Such filter constraints have the form

$$\textsc{filter}(E, b) \equiv \mathrm{EBV}([\![\mu[E]]\!]) = b$$

where μ is the solution mapping that maps each variable $x \in \mathrm{vars}(E)$ to its value. Filter constraints are reified constraints, with b as their truth value. The variable b is a Boolean variable using SPARQL semantics. Hence, its domain contains three states, including the error state.

Considering an implementation computing $\mathrm{EBV}([\![\mu[E]]\!])$ using the semantics described in chapter 2, we have a generic forward-checking propagator for any filter constraint. As soon as all variables but one are assigned, we iterate over all values of the domain of the unassigned variable. Only values for which the evaluation has a truth value that is in the domain of b are kept. The propagator is shown in

```
procedure FilterFC($E, b$)
    if all variables of vars($E$) are assigned then
        $\mu \leftarrow \{ (x, v) \mid x \in \text{vars}(E), v \in D(x) \}$
        Bind($b$, EBV($[\![\mu[E]]\!]$))
    else if only one variable of vars($E$) is unassigned then
        $x_u \leftarrow$ the unassigned variable of vars($E$)
        $\mu \leftarrow \{ (x, v) \mid x \in \text{vars}(E) \setminus \{ x_u \}, v \in D(x) \}$
        ClearMarks($x_u$)
        for all $v \in D(x_u)$ do            ▹ Efficient iteration over $dom_{x_u}[0 \mathrel{..} size_{x_u} - 1]$
            $\mu' \leftarrow \mu \cup \{ (x_u, v) \}$
            if EBV($[\![\mu'[E]]\!]$) $\in D(b)$ then
                Mark($x_u, v$)
            end if
        end for
        Restrict($x_u$)
    end if
end procedure
```

Algorithm 6.6: The generic forward-checking propagator for filter constraints allows Castor to handle any SPARQL filter, provided we can evaluate expressions. The propagator is however much less efficient than specialized propagators.

algorithm 6.6. While this provides a fallback for full SPARQL compliance, we can use more efficient propagators in some cases.

For Boolean operators $(\neg b)$, $(b_1 \land b_2)$ and $(b_1 \lor b_2)$, domain-consistent propagators are easily written. However, special care needs to be taken with the error state. Thanks to the reification of the filter constraints, any Boolean combination of expressions can be propagated in this way.

Comparisons between two variables can also be handled efficiently by specialized propagators. The sameTerm(x, y) filter is true if x and y are assigned the same value, i.e., the same identifier. Such constraint is implemented with value-based domain-consistency. When an identifier is removed from x, it is also removed from y and conversely. If the constraint is false, a forward-checking propagator is used. When one variable is assigned, we remove the assigned identifier from the domain of the other variable.

The equality filter $x = y$ is similar to the sameTerm filter, but we must take into account the fact that identity is not the same as equality in SPARQL. Two values with different identifiers may still be equivalent. To propagate the equality filter, we use a property of the value ordering, stating that equivalent values have neighboring identifiers. For each value, we can retrieve its *equivalence class*. Hence, the equality filter means that x must be an identifier in the equivalence class of y and conversely.

Inequality filters between two variables, like $x < y$, are easily implemented with bound-consistent propagators. As for the equality filter, care must be taken to use the equivalence classes of the values.

For arithmetic expressions, like $x_1 + x_2 = y$, we introduce numerical auxiliary variables. The domains of the auxiliary variables are integer numbers and are implemented with the bound representation. Bound-consistent channeling constraints map RDF variables to numerical auxiliary variables. Bound-consistent propagators for the arithmetic constraints are posted on the auxiliary variables. An integer value i in the domain of a numerical variable is interpreted as the range $[i, i + 1)$, i.e., all integer, floating-point or decimal numbers whose integer part is i. The auxiliary variables thus represent an approximation of the real value. For correctness, the generic forward-checking propagator described above is used. The propagators for arithmetic constraints are used for additional pruning.

6.4 Search

The search tree is defined by using a labeling strategy. At each node, an unassigned decision variable is chosen and a child node is created for each value in the domain of the variable. All the variable selection heuristics presented in section 4.4 have been implemented in Castor. First experiments have not shown much differences between the various standard heuristics (see also section 7.4.1). Hence, research on the search heuristics in Castor has been left for future work. The default variable selection heuristic in Castor is dom/deg, which was marginally better on one query. The ordering of the values is defined by their current order in the dom_x array representation.

The search tree is explored with a depth-first search algorithm. Such exploration is required for efficient inconsistency check of optional subqueries (see section 5.2.1) and efficient restore of the domains. To restore the discrete representation of a domain, we only reset its size. This assumes that removed values, i.e., values above $size_x$ in dom_x, have not changed. Operations on the domain do not change the order of removed values, but do move values that are still in the domain. Restoring a domain, i.e., resetting its size, invalidates any later checkpoints, i.e., whose size is smaller. Hence, only a depth-first search algorithm can be used.

To enable posting constraints during the search, we introduce *subtrees*. A subtree consists of a set of constraints and a set of decision variables to label. Each subtree in Castor corresponds to a BGP of the query. It iterates over all assignments of the variables satisfying the constraints, i.e., the solutions of the BGP, taking into account the variables that were assigned previously. At each solution, Castor can create a new subtree or output the solution, according to the operational semantics described in section 5.2.1. When a subtree has been completely explored, the domains of the variables are restored to their state when the subtree was created and the constraints are removed. The search can then continue in the previous subtree.

6.5 The Castor Open Source System

The system described in the previous sections of this chapter has been implemented as an open-source library.[1] It is written in C++11 and contains about 16,000 lines of code. The library relies on the the *raptor* and *rasqal* libraries [Bec01] to parse the RDF and SPARQL syntax.

This section describes the implementation of the system. Section 6.5.1 shows the working of the library. Section 6.5.2 present the tools shipped with Castor. Finally, section 6.5.3 discusses some limitations of the system.

6.5.1 The Library

The main classes of the library are `Store` and `Query`. The `Store` class contains all the operations for reading the on-disk dataset presented in section 6.1. The `Query` class implements query parsing and evaluation. An instance of the `Query` class is bound to a specific query and is discarded after execution. An instance of the `Store` class may be reused for different queries, thus keeping the cache of decompressed triple pages. Currently, Castor does not allow concurrent access to the store.

Because a dataset can contain a huge number of values, the domain initialization can take some time. To avoid performing the initialization again for every query, the store keeps a pool of domains. When creating a query, domains are requested from the store. They are returned to the store when the query is done. Such optimization is possible because the domain initialization depends solely on the dataset and not on the query to be solved.

Listing 6.1 shows an example usage of the Castor library. Lines 2 and 3 respectively create the store and the query. The query creation does not start the CP search. Lines 4–12 iterate over the solutions and print them out. Each time the `next()` function is called, a portion of the search space is explored. If a solution is found, the execution is halted and `next()` returns `true`. The next call to the function resumes the search, starting from that solution. When the whole search space is explored, `next()` returns `false`.

The call to `ensureDirectStrings` on line 9 is needed to print out the value. Remember that strings in values are represented by identifiers. Value look-ups happen often when evaluating SPARQL expressions. In such cases, string identifiers often suffice. Looking up the content of the string would add an unneeded cost. Hence, string identifiers are not resolved automatically when looking up values.

6.5.2 Tools

Along with the library, Castor provides a set of tools. The `castorld` program ("`ld`" stands for "load") transforms an RDF dataset from any syntax understood by *raptor* to the Castor representation explained in section 6.1. It is responsible for sorting

[1] available on `https://github.com/vianney/castor`

```
int main(int argc, char* argv[]) {
    castor::Store store("dataset.db");
    castor::Query query(&store, "SELECT * WHERE { ?s ?p ?o }");
    while(query.next()) {
        cout << "Solution:" << endl;
        for(int i = 0; i < query.requested(); i++) {
            castor::Variable* x = query.variable(i);
            castor::Value val = store.lookupValue(x->valueId());
            val.ensureDirectStrings(store);
            cout << "  " << x->name() << ": " << val << endl;
        }
    }
    cout << "Found " << query.count() << " solutions." << endl;
    return 0;
}
```

Listing 6.1: Example code solving a simple query with the Castor library. The store may be reused for other queries.

the strings, the values and the triples and for constructing the B$^+$-trees. As the dataset may not fit entirely in main memory, an external sort algorithm is used. Thus, the memory consumption of `castorld` is constant with respect to the dataset size, assuming *raptor* can read the input syntax incrementally.

The `castor` and `castord` programs solve SPARQL queries on a dataset generated by `castorld`. The `castor` tool executes a single query, received as an argument, and outputs the solutions on the standard output. The `castord` tool ("`d`" stands for "daemon") implements a SPARQL *endpoint*. It launches a basic HTTP server answering queries following the SPARQL Protocol [CFT08]. Note that this server is not production-ready. A query will block the whole server until completion. No limitations, e.g., a timeout or a limit on the number of results, are enforced, and concurrent queries are not supported.

6.5.3 Limitations

Castor is a research prototype. It is not yet suitable for real-world applications. In this section, we discuss some limitations of the current implementation.

Castor targets the SPARQL 1.0 specification [PS08]. The following parts are however not yet implemented.

- Named graphs. A SPARQL query can be performed over multiple datasets. A GRAPH operator allows to restrict part of the query to one of the datasets. Such functionality can be implemented by extending our triple store into a *quad* store, i.e., storing (g, s, p, o) quads instead of (s, p, o) triples. The g component corresponds to the IRI of the originating graph.

- CONSTRUCT queries. The results of a SELECT query can be easily transformed into an RDF graph with the CONSTRUCT template.
- DESCRIBE queries. Like for the CONSTRUCT query form, a DESCRIBE query returns an RDF graph instead of a solution mapping. The graph should contain a description of the result values. The precise definition of the description is not specified.
- Casting operators. In a SPARQL filter, values can be interpreted as another type. For example, `xsd:integer("42")` will evaluate to the numerical literal 42. Such operators should not be difficult to add to the generic forward-checking propagator.

Thanks to the generic forward-checking propagator, most SPARQL filters are supported. However, the generic propagator is not efficient. Specialized propagators exist for arithmetic constraints, variable-to-variable comparisons and variable-to-value comparisons. More complex filters will likely be slow due to the generic propagator.

The dataset representation is read-only. To add a triple to the dataset, the entire on-disk structure must be recreated. Castor relies on the ordering of values and triples to efficiently retrieve data of interest. Adding a value or a triple involves rebuilding the B^+-trees. One possible solution is to use deferring indexes [NW10], keeping updates in additional trees. Such trees are then periodically merged with the main indexes when the server is idle.

Chapter 7

Evaluation

In this chapter, we evaluate the performances of Castor. We first present the benchmarks that will be used. Then, we compare Castor with state-of-the-art engines. The third section briefly compares Castor with the Comet CP solver. The last section covers some technical design choices made by Castor.

7.1 Benchmarks

In order to evaluate the performances of Castor, we use standard benchmarks. The SPARQL Performance Benchmark, presented in section 7.1.1, is especially interesting as it contains difficult queries. Even state-of-the-art engines have difficulties to solve such queries. The Berlin SPARQL Benchmark, shown in section 7.1.2, is used to assess the scalability of the engines. The queries are simpler, but the datasets are larger.

7.1.1 SPARQL Performance Benchmark

The SPARQL Performance Benchmark (SPPB) [Sch+09] is modeled after the DBLP database [Ley13]. The DBLP database contains metadata about academic publications, including their authors, the publishing journal, etc. The social-world distribution of the DBLP database, i.e., most nodes have a low degree, captures the social network aspect of the Semantic Web well. Indeed, the Semantic Web is built by combining many small datasets.

The SPPB includes a deterministic generator that produces DBLP-like datasets of arbitrary size. The benchmark uses sizes ranging from 10,000 to 5 million triples. The values over triples ratio is roughly 0.6. Thus, for the 5 million triples dataset, the initial domain of each variable contains 3 million values or so.

The benchmark provides 17 hand-made queries, designed to cover many use cases. Some queries are variants of other queries. They are suffixed with a small letter. Table 7.1 shows an overview of the queries and their general form.

Query	Description
S1	Single BGP with one result
S2	P_1 OPT P_2, where P_2 is a single triple pattern
S3a	P FILTER $(x = c)$ with many results
S3b	P FILTER $(x = c)$ with few results
S3c	P FILTER $(x = c)$ with no results
S4	P FILTER $(x < y)$ with a very large number of results
S5a	P FILTER $(x = y)$
S5b	Single BGP, same results as S5a
S6	P_1 DIFF (P_2 FILTER $(x_1 = x_2 \wedge y_1 < y_2)$)
S7	P_1 DIFF (P_2 DIFF P_3)
S8	P_1 AND ((P_2 FILTER E_1) UNION (P_3 FILTER E_2))
S9	P_1 UNION P_2 with unbound predicates
S10	Single triple pattern with unbound predicate
S11	Single triple pattern with LIMIT and ORDER modifiers
S12a	ASK version of S5a
S12b	ASK version of S8
S12c	ASK query of a single triple not present in the dataset

Table 7.1: The queries provided by the SPARQL Performance Benchmark cover a wide range of use cases. The authors of the benchmark have identified queries S4, S5a, S6 and S7 as being the most challenging for current SPARQL engines. The complete queries are included in appendix A.1.

```
SELECT DISTINCT ?person ?name
WHERE {
  ?art rdf:type bench:Article .
  ?art dc:creator ?person .
  ?inproc rdf:type
    bench:Inproceedings .
  ?inproc dc:creator ?person2 .
  ?person foaf:name ?name .
  ?person2 foaf:name ?name2
  FILTER (?name=?name2)
}
```

(a) Query S5a

```
SELECT DISTINCT ?person ?name
WHERE {
  ?art rdf:type bench:Article .
  ?art dc:creator ?person .
  ?inproc rdf:type
    bench:Inproceedings .
  ?inproc dc:creator ?person .
  ?person foaf:name ?name
}
```

(b) Query S5b

Listing 7.1: Query S5a returns the names of all persons that occur as author of at least one inproceeding and at least one article. The filter involving two variables is challenging for SPARQL engines. Query S5b computes the same set of results without the filter.

Query S1 is a simple basic graph pattern with only one result. With efficient indexes, the execution time is expected to be constant with respect to the dataset size.

Query S2 has an optional pattern consisting of a single triple pattern with a shared variable. For every solution of P_1, there is at most one solution for P_2.

Queries S3a, S3b and S3c are small basic graph patterns with a filter assigning a constant value to a variable. The queries differ only by the constant value appearing in the filter.

Query S4 has the particularity of computing a very large result set, two orders of magnitude larger than the other queries.

Query S5a, shown in listing 7.1, is a basic graph pattern with an equality filter between two variables. As shown in chapter 2, the equivalence between two values does not always mean that those two values are identical. However, in this particular dataset, there is a one-to-one mapping between the persons and their names. Query S5a can thus be rewritten as query S5b, without a filter.

Queries S6 and S7 make use of negations. The filters in S6 always involve a variable of P_1 and a variable of P_2.

Query S8 involves an union pattern with filters. The filters are all of the form $x \neq y$.

Queries S9 and S10 involve unbound predicates, i.e., triple patterns with a variable as predicate. SPARQL engines tend to be less optimized for such less common queries.

Query S11 returns the top-n results of a query by combining an ORDER and a LIMIT solution modifier.

Queries S12a, S12b and S12c are ASK queries, i.e., limited to the first result found.

The authors of the benchmark have found that queries S4, S5a, S6 and S7 are the most challenging ones for current SPARQL engines. Those queries involve filters with more than one variable and/or DIFF patterns. Engines based on relational database technology usually post-process filters. Constraint programming, however, can exploit filters during the search.

The benchmark procedure is to run each query on a freshly started store. The system cache should be cleared between each query. Such procedure is motivated because the queries are not randomized. An engine could cache the results for a query and serve it back when called again. Measuring such behavior would not be relevant.

7.1.2 Berlin SPARQL Benchmark

The Berlin SPARQL Benchmark (BSBM) [BS09] is described as follows on its website[1].

> The Berlin SPARQL Benchmark (BSBM) defines a suite of benchmarks for comparing the performance of [SPARQL endpoints] across architectures. The benchmark is built around an e-commerce use case in which a set of products is offered by different vendors and consumers have posted reviews about products. The benchmark query mix illustrates the search and navigation pattern of a consumer looking for a product.

The BSBM includes a deterministic data generator that produces datasets of arbitrary size, specified by a scale factor. The experiments conducted by the authors of the benchmark use datasets with 25 million, 50 million, 100 million, and 200 million triples. The values over triples ratio is about ¼. Thus, the 100 million triples dataset contains 25 million values or so.

The BSBM covers three use cases.

- The Explore use case simulates a consumer looking for a product.
- The Explore and Update use case simulates a read/write scenario using SPARQL 1.1 Update queries.
- The Business Intelligence use case rely on SPARQL 1.1 features such as grouping and aggregation.

Because Castor does not support the additional SPARQL 1.1 features, we focus on the Explore use case.

The Explore use case contains 12 SPARQL query patterns to be instantiated by the test driver, by replacing placeholders with random values. The test driver sends queries successively to the tested SPARQL engine and measures the total time taken for the whole query mix, as well as the execution time of each query.

[1]`http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/`

Query	Description	DISTINCT	ORDER + LIMIT
B1	P FILTER $(x > c)$	✓	✓
B2	BGP with simple OPT patterns		
B3	DIFF pattern		✓
B4	UNION pattern	✓	✓
B5	BGP with arithmetic constraints	✓	✓
B7	Complex OPT patterns		
B8	langMatches filter		✓
B11	Unbound predicates		

Table 7.2: The queries provided by the Berlin SPARQL Benchmark are similar to relational database access. The complete queries are given in appendix A.2.

In contrast to the SPPB, the BSBM is designed to evaluate engines with a warm cache. It does so by first running 2000 warm-up query mixes without measuring the performances. Then it performs the real benchmark with 500 query mixes. The procedure is sound because the queries are randomized. Measuring warm-cache performance gives a better insight on real-world performances than cold-cache performances. In real-world applications, SPARQL engines are often long-lasting servers.

The queries are very similar to relational database accesses. Table 7.2 shows an overview of the queries. Four queries out of 12 were left out:

- Query B6 has a regular expression filter and is deprecated by the authors of the benchmark.
- Queries B9 and B12 respectively use the DESCRIBE and CONSTRUCT query forms.
- Query B10 uses a casting operator in a SPARQL expression.

Such features are unsupported by Castor (see section 6.5.3).

As shown in table 7.2, most queries include an ORDER and a LIMIT solution modifier. In such queries, the highest limit is 20 solutions. A simple OPT pattern consists of a single triple pattern with only one non-shared variable.

Query B1 searches for products with specific features. Hence, it touches a lot of data across the whole RDF graph. Query B2 gathers information about a single product. The evaluation only covers a small subgraph of the dataset.

Query B5 is interesting as it uses arithmetic constraints. It is shown in listing 7.2. The arithmetic constraints can be efficiently exploited by Castor during the search.

7.2 SPARQL Engine Comparison

To evaluate the performances of Castor, we compare against state-of-the-art SPARQL engines on the benchmarks described above. In the first part of the section, we present the contenders. Then, we show the results of the SPPB and the BSBM benchmarks.

```
SELECT DISTINCT ?product ?productLabel
WHERE {
  ?product rdfs:label ?productLabel .
  FILTER (%ProductXYZ% != ?product)
  %ProductXYZ% bsbm:productFeature ?prodFeature .
  ?product bsbm:productFeature ?prodFeature .
  %ProductXYZ% bsbm:productPropertyNumeric1 ?origProperty1 .
  ?product bsbm:productPropertyNumeric1 ?simProperty1 .
  FILTER (?simProperty1 < (?origProperty1 + 120) &&
          ?simProperty1 > (?origProperty1 - 120))
  %ProductXYZ% bsbm:productPropertyNumeric2 ?origProperty2 .
  ?product bsbm:productPropertyNumeric2 ?simProperty2 .
  FILTER (?simProperty2 < (?origProperty2 + 170) &&
          ?simProperty2 > (?origProperty2 - 170))
}
ORDER BY ?productLabel
LIMIT 5
```

Listing 7.2: Query B5 searches for products that are similar to another one. The arithmetic filter can be efficiently exploited in constraint programming.

7.2.1 Considered SPARQL Engines

We consider three well-known open-source SPARQL engine.

Sesame [BKH02] is a modular Java API to access the Semantic Web, developed by Aduna. It includes an in-memory and an on-disk triple store capable of solving SPARQL queries. The store can be swapped with another implementation by using plugins, such as Ontotext's proprietary OWLIM. In this chapter, we use the built-in on-disk triple store.

Sesame is not known for stellar performances, but it gives a baseline of the performances of common non-optimized engines. We use Sesame 2.6.1 running on Java 1.6.0 update 30. The native on-disk store is configured with 3 indexes: SPOG, POSG, and OSPG.

4store [HLS09] is an efficient SPARQL engine written in C, developed at Garlik with a focus on scalability. It is designed to operate in large clusters on datasets with billions of triples. We use 4store 1.1.5 with the default configuration.

Virtuoso [EM09] is a large database system written in C, developed by OpenLink Software. It is both a relational database system and an RDF triple store. The main DBPedia servers are powered by Virtuoso. We use the open-source edition of Virtuoso, version 6.1.6, with the default configuration.

Virtuoso is known to have good performances due to aggressive optimization. However, such optimization does not always respect the SPARQL specification. For example, Virtuoso does not distinguish between the values (`"1"`,`xsd:integer`) and (`"1.0"`,`xsd:float`).

Even though Castor borrows some concepts from RDF-3X, we do not compare with it. RDF-3X implements only basic graph patterns and does not handle filters.

7.2.2 SPARQL Performance Benchmark

We have generated 5 datasets with 10k, 25k, 250k, 1M and 5M triples. We have performed cold runs of each query over all the generated datasets, i.e., between two runs the engines were restarted and the system caches were cleared with "`sysctl -w vm.drop_caches=3`". We have set a timeout of 30 minutes.

Each query is run three times for each dataset to exclude variations incurred by the operating system. We have observed no significant variance in query execution time.

Note that cold runs may not give the most significant results for some engines. E.g., Virtuoso aggressively fills its cache on the first query in order to perform better on subsequent queries. However, such setting is required by the non-randomized queries.

All experiments were conducted on an Intel Pentium 4 3.2 GHz computer running ArchLinux 64bits with kernel 3.8.8, 3 GB of DDR-400 RAM and a 40 GB Samsung SP0411C SATA/150 disk with ext4 filesystem. We report the time spent to execute the queries, not including the time needed to load the datasets. The mean execution times are shown in fig. 7.1. We do not show the standard deviation of the measurements, as it was negligible for all instances.

Exploiting the constraints during the search gives an advantage to Castor. Queries S5a and S5b compute the same set of solutions. S5a enforces the equality of two variables with a filter, whereas S5b uses a single variable for both. Note that such optimization is difficult to do automatically, as equivalence does not imply identity in SPARQL (see chapter 2). Detecting whether one can replace the two equivalent variables by a single one requires a costly analysis of the dataset, which is not performed by any of the tested engines. Sesame and 4store timed out when trying to solve query S5a on the 250k and above datasets. Because Virtuoso breaks the SPARQL standard and treats equality as identity, it performs as well on both queries S5a and S5b. Castor does no query optimization, but still performs equally well on both variants thanks to its ability to exploit the filter at every node of the search tree.

Query S12a replaces the SELECT query form by ASK in S5a. The solution is a boolean value reflecting whether there exists a solution to the query. Thus, we only have to look for the first solution. However, Castor still needs to initialize the search tree, which is the greatest cost. A similar behavior is observed in query S1. Sesame and 4store show a near-constant execution time, while Castor has to go through the entire CP process.

Executing query S4 results in many solutions (e.g., for the 1M dataset, S4 results in 2.5×10^6 solutions versus 3.5×10^4 solutions for S5a). The filter does not allow for

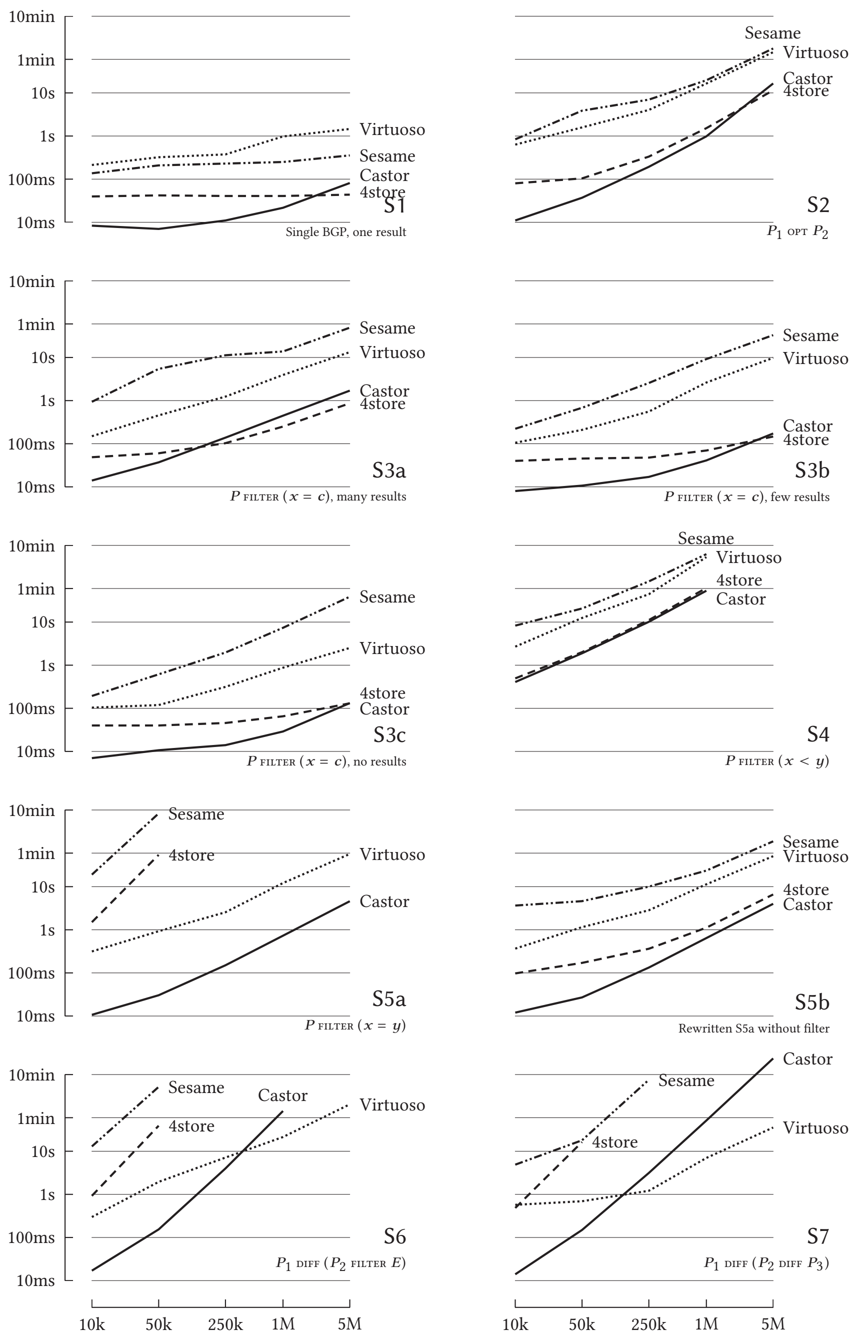


Figure 7.1: Castor outperforms state-of-the-art engines on queries from SPPB with filters. It is competitive on other queries. The x-axis shows the dataset sizes. The y-axis shows the query execution time. Note that both axes have a logarithmic scale.

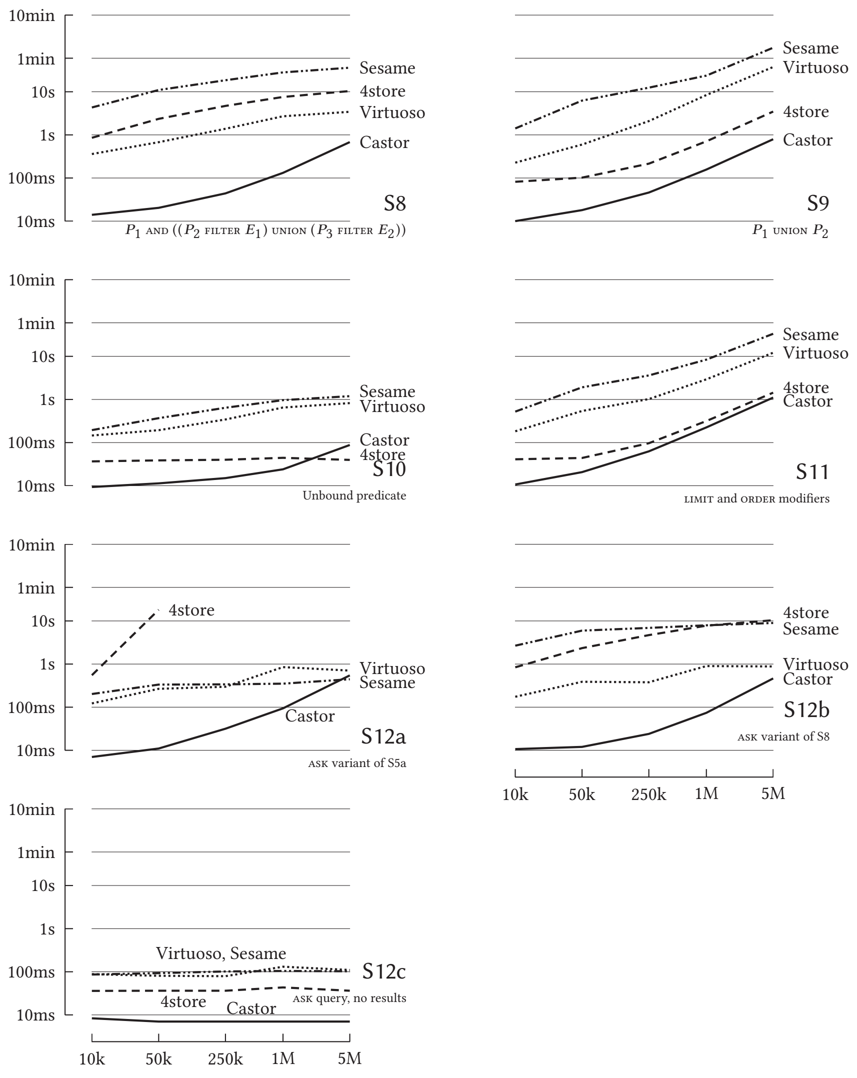


Figure 7.2: Comparison of SPARQL engines on the SPPB benchmark (continued)

much pruning. Nevertheless, Castor is still competitive with the other engines. None of the engines were able to solve the query for the 5M dataset in less than 30 minutes.

Queries with negations, i.e., S6 and S7, are not handled well by Castor, nor by Sesame or 4store. Only Virtuoso is able to return results in a reasonable amount of time.

On other queries, Castor is very competitive with state-of-the-art engines. However, the execution time increases quicker than for other engines (see for example queries S2, S8, S10 and S12b). This observation leads to the question whether the constraint programming approach scales to large datasets. Such question is investigated in the next section.

7.2.3 Berlin SPARQL Benchmark

We have generated 4 datasets with 25M, 50M, 100M, and 200M triples. The provided test driver is used to run 2000 warm-up query mixes and 500 benchmark query mixes. For each combination of dataset and engine, we followed the following procedure.

1. Start the server.
2. Create a new database and import the RDF graph.
3. Shutdown the server, clear caches with "`sysctl -w vm.drop_caches=3`", and restart the server.
4. Launch the BSBM test driver.

All experiments were conducted on a KVM virtual machine running on an AMD Opteron 6284 2.7 GHz computer with ArchLinux 64bits, kernel 3.9.3, 16 GB of main memory, and 500 GB of disk space with ext4 filesystem. The virtual disk partition was directly mapped to an LVM volume, using the virtio driver, providing low virtualization overhead. One core was allocated to the virtual machine. To make better use of the available RAM, Castor was configured to keep a cache of 50,000 triple pages, amounting to 9.2 GB. We report the average time spent to execute the queries, not including the time needed to load the datasets. The mean execution times are shown in fig. 7.3.

Overall, Castor is competitive with state-of-the-art engines on these larger graphs. As expected, Castor is able to outperform other engines on query B5 by efficiently exploiting the arithmetic constraints. Surprisingly, Castor is also able to best Virtuoso and 4store on query B11. Query B11 is a UNION of two single triple patterns with unbound predicates. We suspect the performances of Castor can be attributed to the more complete RDF-3X indexes.

7.3 The Need for a Specialized Solver

The model described in chapter 5 can be used to solve SPARQL queries with an off-the-shelf CP solver. However, the huge domains and large triple table make their use

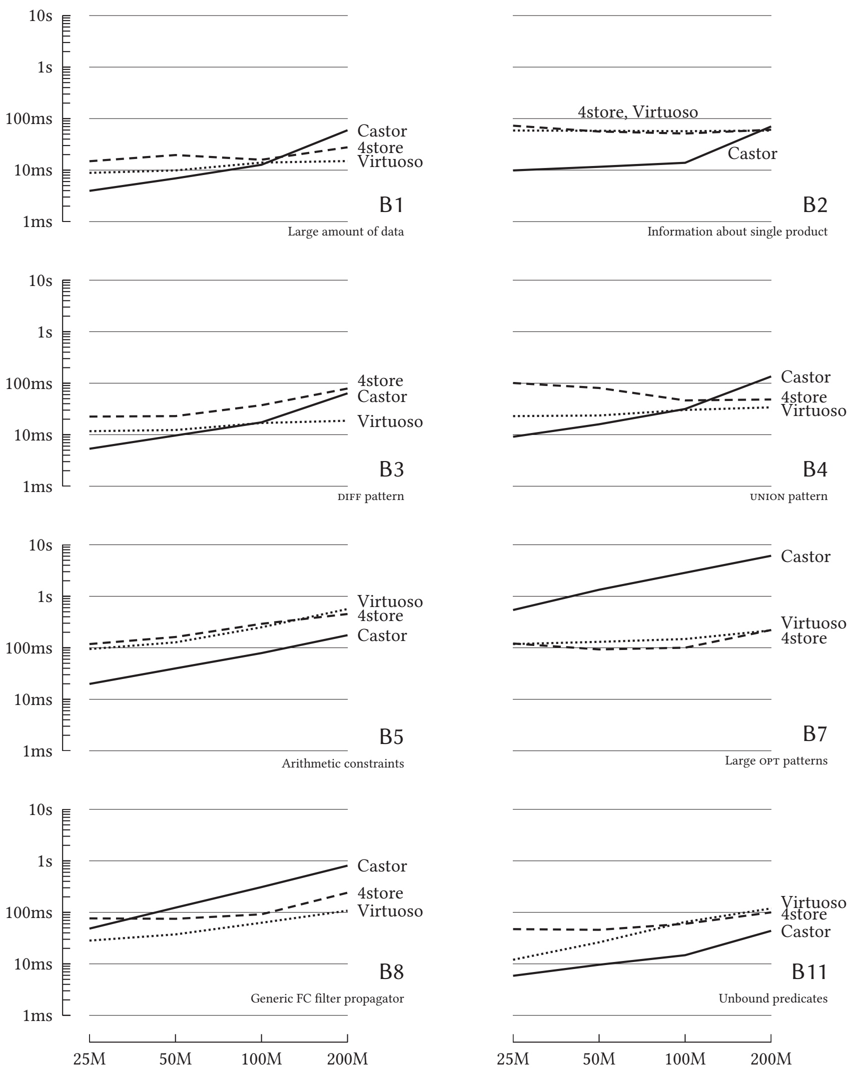


Figure 7.3: Castor is competitive with state-of-the-art engines on large datasets from the BSBM. On queries with complex filters, e.g., B5, Castor keeps a distinct advantage. The x-axis shows the dataset size in number of triples. The y-axis shows the average query execution time. Both axes have a logarithmic scale.

impractical.

To show this point, we have implemented the model in Comet [Dyn10]. Values are mapped to integer identifiers as described in section 6.1.1. The built-in table constraint is used. The entire triple table is loaded in memory. The loading itself is not included in the execution time. Built-in constraints are used for the filters. Such constraints do not respect the SPARQL specification entirely, but nevertheless return correct results in our particular test settings. The built-in `labelFF` search strategy is used.

We have run a subset of the SPPB benchmark, including the challenging queries. As we have not implemented solution modifiers in our Comet model, we have modified the queries to remove any solution modifier.

The Comet implementation performs very badly on all queries as shown in fig. 7.4. The results are worse than state-of-the-art SPARQL engines on all queries, except S5a′, where Comet efficiently exploits the equality filter during the search.

A specialized solver is thus needed in Castor in order to compete with state-of-the-art SPARQL engines.

7.4 Impact of Technical Choices

Chapter 6 presented alternative design choices of some aspects. In this section, we evaluate the various alternatives and show the rationale behind the default choices made by Castor.

The following subsections deal with the search heuristics, the propagation of the triple constraints, the propagation of the filter constraints and the size of the triple cache.

7.4.1 Search Heuristics

To evaluate the impact of the variable selection heuristic on the query execution time, we have run the SPPB benchmark with the following heuristics.

- dom: select the variable with the smallest domain,
- deg: select the variable with the highest degree,
- ddeg: select the variable with the highest dynamic degree,
- dom/deg: select the variable with the smallest domain size over degree ratio,
- dom/ddeg: select the variable with the smallest domain size over dynamic degree ratio.
- random: select an unassigned variable at random. Results for the random heuristic are averaged over 10 runs.

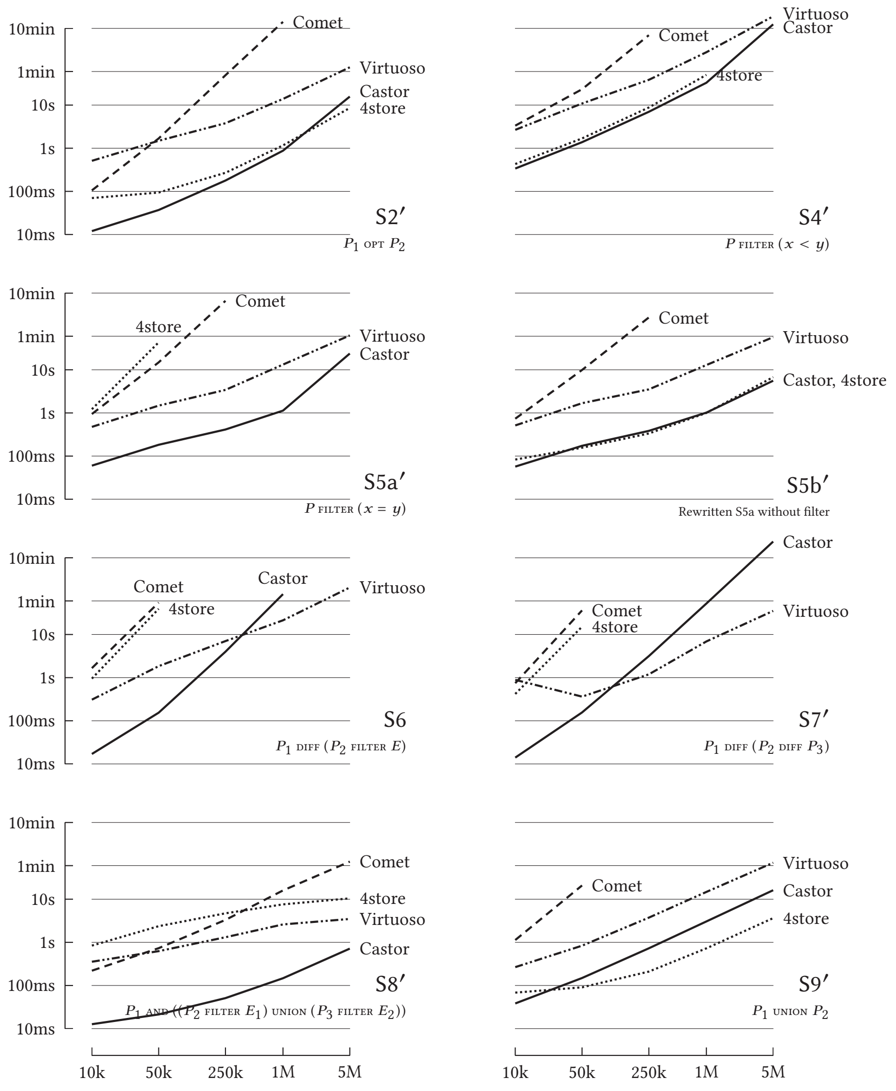


Figure 7.4: The off-the-shelf Comet solver does not perform well on any query. It is however able to best 4store on query S5a′ by exploiting the equality filter. Note that all solution modifiers have been removed from the queries.

The deg heuristic is static, i.e., the ordering of the variables does not change during the search. The other heuristics are dynamic.

The search for a solution of query S1, S3b, S3c, S10, S12a, S12b, or S12c, is (mostly) backtrack-free. Hence, the chosen search heuristic has no impact on the performances of solving the query. The results for the other queries are shown in fig. 7.5. Generally, all search heuristics perform equally well, with a slight advantage for the dom/deg heuristic. It is thus the default heuristic in Castor.

Heuristics based on variable degree alone have a slight disadvantage. Considering the small number of constraints involved in a query, this is not surprising. The degrees are not different enough to differentiate between the variables. On the other hand, the domain size gives an effective measure. Due to the large number of values in the dataset, the domain size can vary widely from one variable to another.

Query S11 has only two variables and only one triple, involving both variables. As the ORDER and LIMIT solution modifiers are used together, the branch-and-bound technique described in section 5.2.2 is used. The variable to be minimized has degree 2, because it is also involved in the bounding constraint. Hence, it is chosen first by the variable selection heuristics deg and dom/deg, which obviously is the right choice.

7.4.2 Consistency Level of the Triple Constraint

Section 6.3.1 presented three different propagators for the triple pattern constraint. To evaluate which consistency level should be achieved, we have run the SPPB benchmark using the following three propagators.

- FC: simple forward checking,
- FC+: forward checking with one-time domain consistency when only two variables are unassigned,
- DC: full domain consistency.

The FC+ propagator is always better than the DC propagator, as shown in fig. 7.6. This is expected, as the DC propagator is called on each variable modification and has to traverse the whole table. The FC+ propagator only achieves domain consistency once, when one variable is bound.

Compared to the simpler FC propagator, the FC+ propagator performs better in the challenging queries S5a and S7, as well in queries S5b, S9, S10 and S11. The additional propagation performed higher up in the tree (when two variables are unassigned) allows FC+ to prune larger parts of the search tree. Triple patterns in SPARQL queries often have at least one constant component. Hence, the FC+ propagator can often achieve its one-time domain consistency at the very beginning of the search.

The FC propagator performs slightly better on ASK queries where we stop after the first solution found. The additional pruning performed by the FC+ propagator

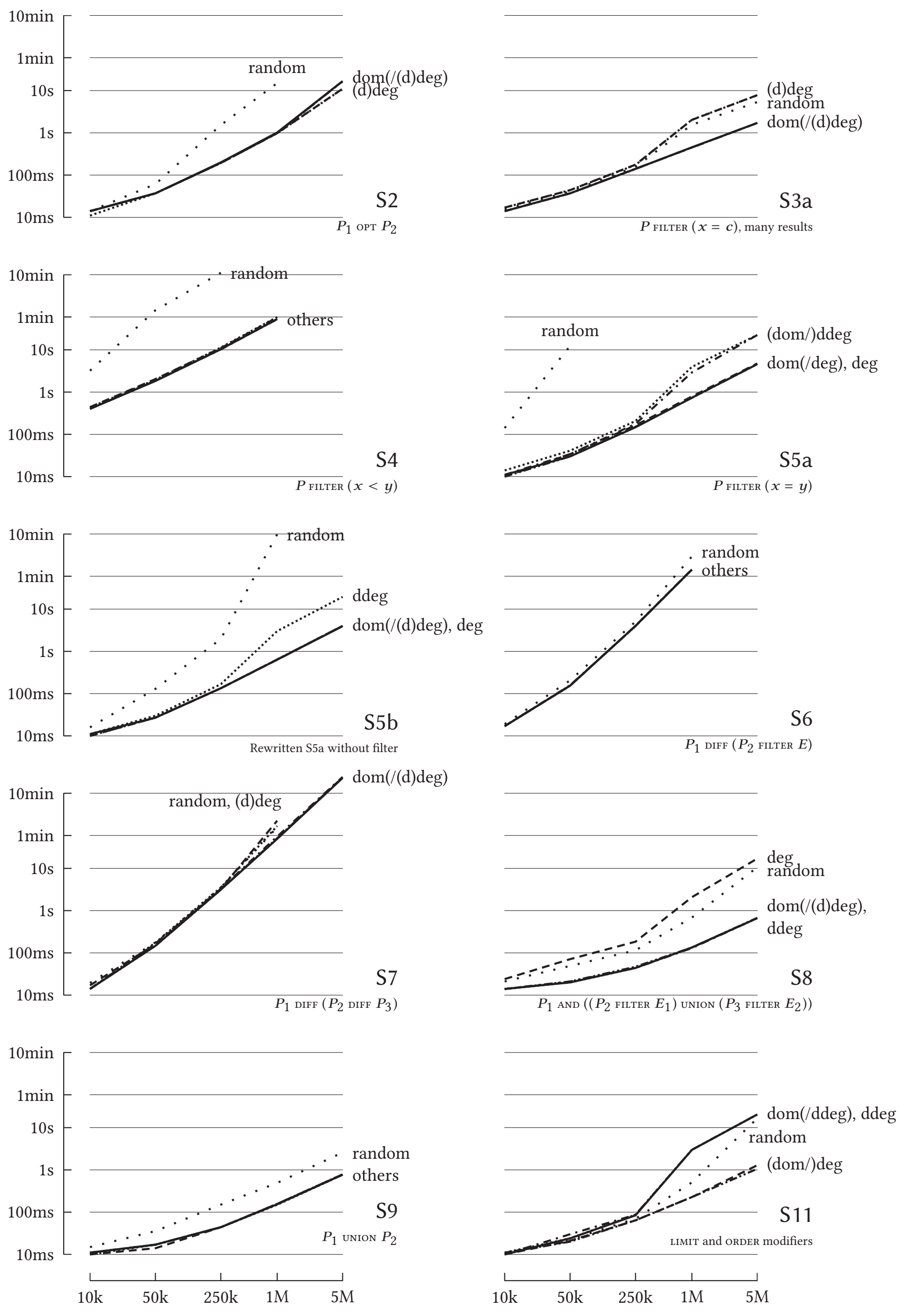


Figure 7.5: The dom/deg search heuristic is the overall winner. Only queries where the execution time differs between the search heuristics are shown.

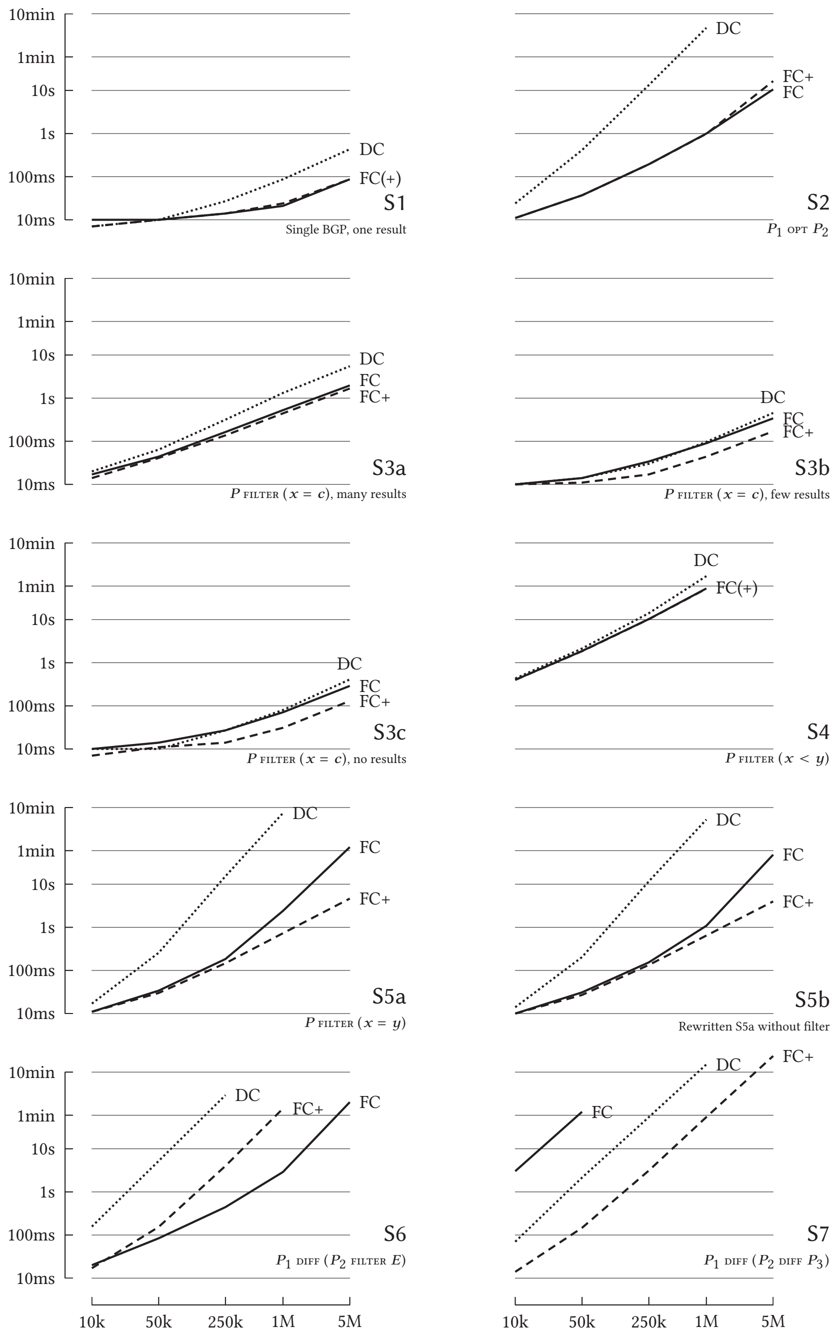


Figure 7.6: The FC+ propagator for the triple constraint is the overall best-performing. However, the FC propagator is clearly better on the challenging query S6.

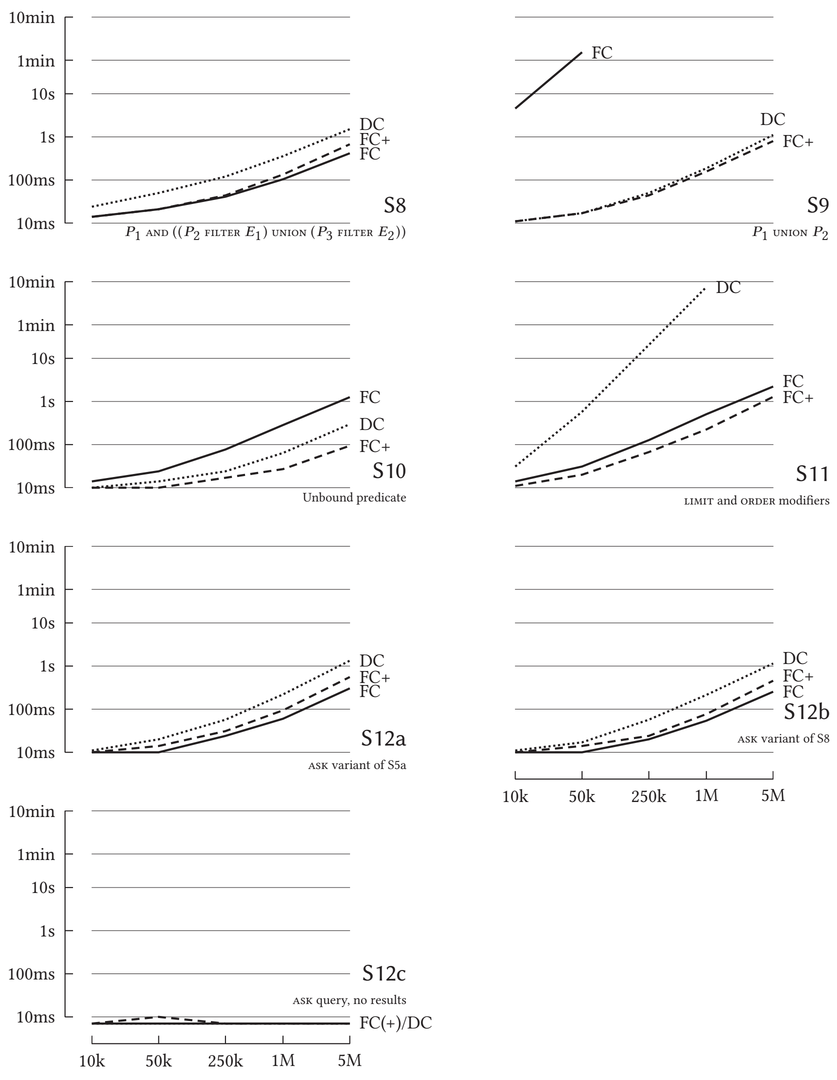


Figure 7.7: Comparison of propagators for the triple constraint (continued)

is unnecessary if the solution can be reached without too many backtracks. Such behavior also appears in the challenging query S6. When evaluating the P_1 DIFF P_2 pattern, we only want to know whether P_2 is satisfiable. Hence, we also stop after the first solution. While we gain some performances by using FC if there is a solution, we may also loose efficiency if there are no solutions. In such cases, the whole search (sub)tree must be explored. The FC+ propagator is then more efficient, as shown by the results of query S7.

No easy rule could be found for choosing between FC and FC+. The FC+ propagator is however more efficient than FC in most cases. Hence, it was chosen as default propagator in Castor.

7.4.3 Propagation of Filter Constraints

Exploiting filters during the search is one of the main interests of using constraint programming to solve SPARQL queries. To assess such claim, we have run the SPPB benchmark with three configurations of Castor.

- Post-process: all filters are post-processed as current SPARQL engines without optimization would do. In this configuration, the P_1 DIFF P_2 pattern is handled as a $(P_1$ OPT $P_2)$ FILTER $(\neg \mathsf{bound}(x))$ pattern, where the filter is post-processed.
- FC: the DIFF pattern is handled as explained in chapter 5. All other filters are enforced using the generic forward-checking propagator shown in section 6.3.2.
- Specialized: filters are enforced using specialized propagators. Equality constraints achieve domain consistency. Inequality constraints achieve bound consistency.

The configuration with specialized propagators outperforms the other configurations on all queries with filters, as shown in fig. 7.8. On challenging queries like S5a and S6, having specialized propagators is especially important. In such queries, the specialized propagators achieve a higher consistency level than the generic FC propagator. Hence, more pruning is performed higher up the search tree.

Note that the generic forward checking propagator has a high cost due to the traversing of the whole domain of the unbound variable. It is sometimes better to post-process the filters, as demonstrated by the challenging queries S4 and S6. Both queries have inequality filters $x < y$. We speculate that the cost of performing the generic forward checking propagator on such filters is too high compared to the few pruning gained.

7.4.4 Impact of the Triple Cache

The propagation of the triple constraints accesses the triple store. Such propagation is performed many times during the evaluation of a query. Leaf pages in the triple store are compressed. To avoid decompressing the same leaf pages over and over, Castor maintains a cache of recently decompressed leaf pages.

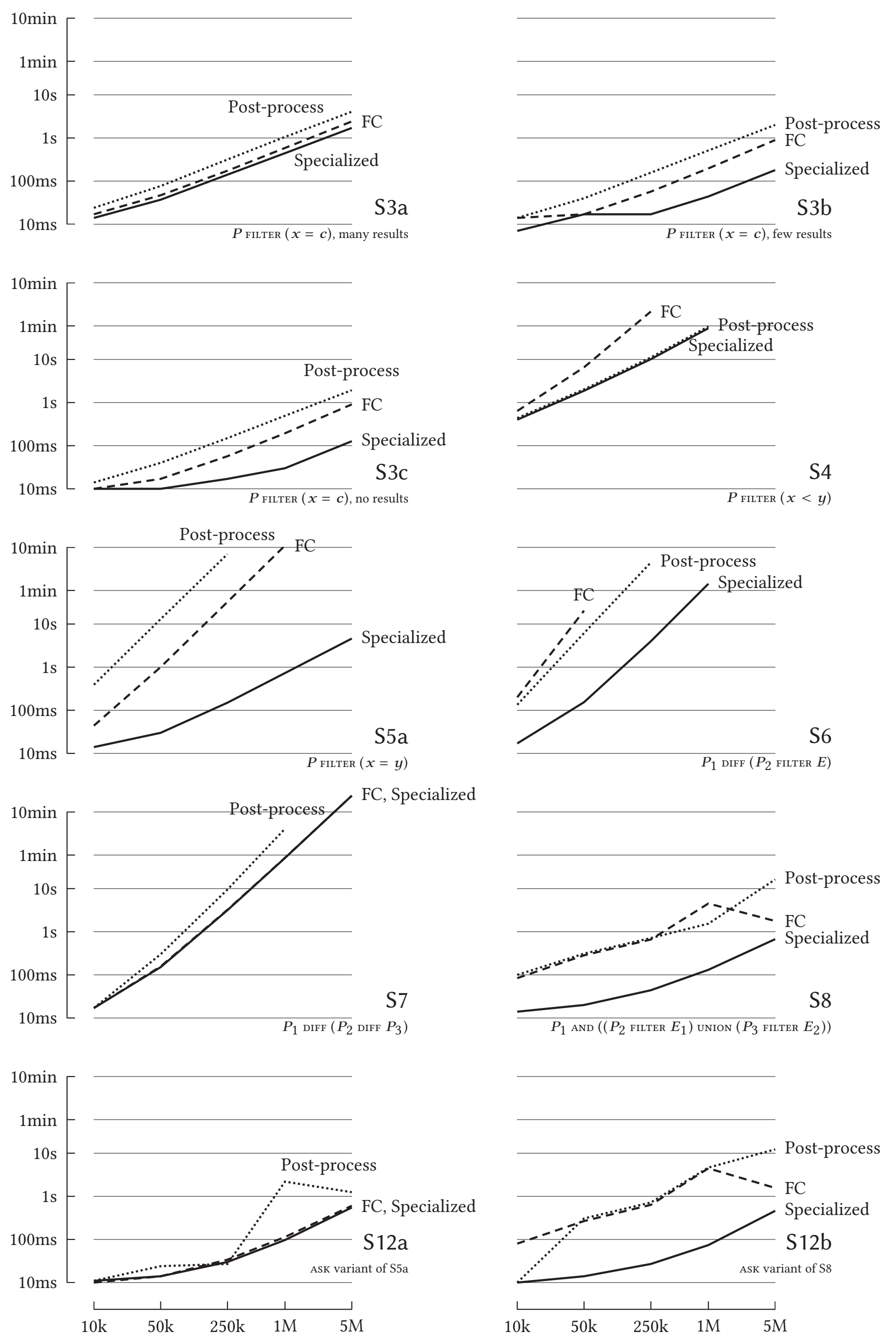


Figure 7.8: Specialized filters improve on all queries. Using the generic forward-checking propagator (FC) is usually better than post-processing the filter (Post-process). However, there are cases where traversing the whole domain is too costly, e.g., queries S4 and S6.

To evaluate the impact of the cache size, we have run the BSBM benchmark on the 100M dataset with varying sizes. The results in fig. 7.9 show little impact of the cache size on the query execution time. As expected, the performances are slightly improving with higher cache sizes.

However, with very large sizes, i.e., 50k and 100k, the query execution times are slightly increasing again. Higher cache size implies higher memory usage. Hence, less memory is left for the system cache and disk reads are slower.

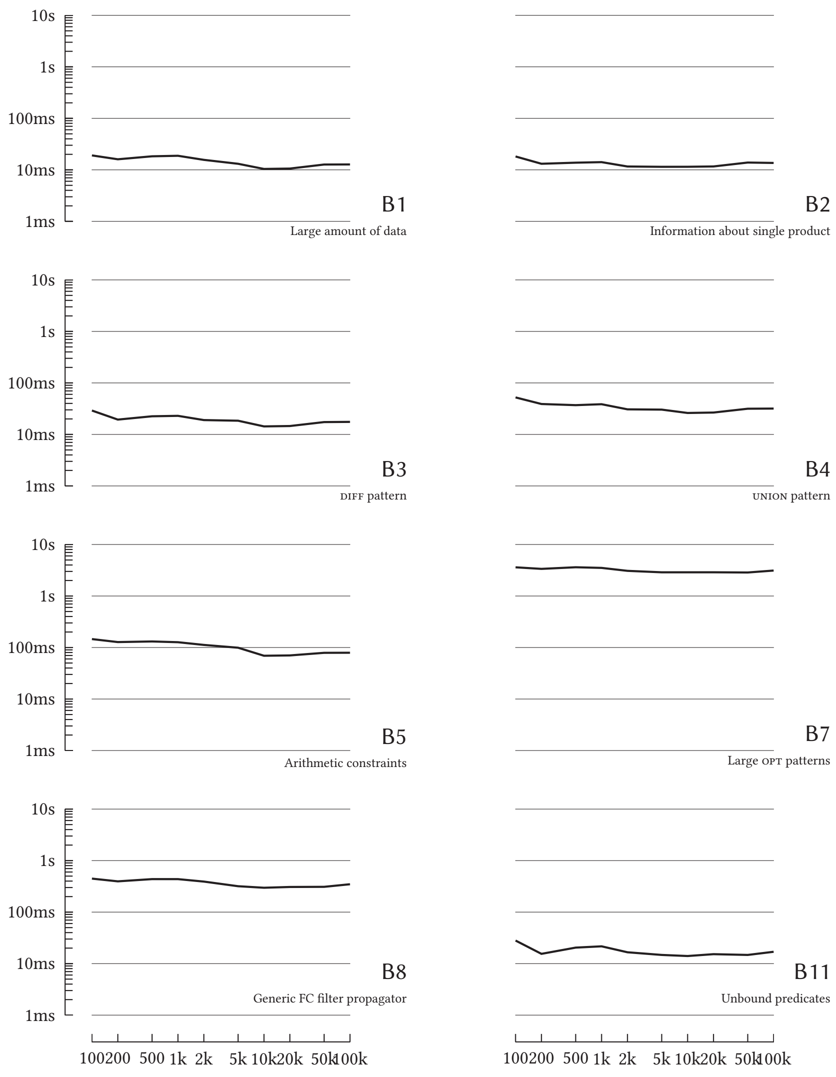


Figure 7.9: The size of the triple cache does not have a high impact on the query execution time. The graphs show the average query execution time of Castor on the 100M dataset. The x-axis represents the size of the triple cache.

Chapter 8

Conclusion

The goal of this thesis is to evaluate the feasibility and efficiency of solving SPARQL queries using Constraint Programming (CP). To this effect, we proposed a reformulation of SPARQL semantics by means of CSPs, and an operational semantics that can be implemented in off-the-shelf CP solvers. We also introduced Castor, a SPARQL engine implementing our semantics with a lightweight specialized solver. Section 8.1 recalls the main achievements of this thesis. Section 8.2 outlines the limitations of our approach, as well as possible directions for future research.

8.1 Results

The first step in solving SPARQL queries with CP consists in reformulating the semantics by means of Constraint Satisfaction Problems (CSPs). This is done in chapter 5. A basic graph patterns (BGP) is translated directly into a CSP with table constraints. Filters on BGPs are added to the set of constraints, and are thus exploited during the search.

Compound patterns cannot always be translated to pure CSPs. Unlike CSPs, not all variables need to be assigned in the solutions of SPARQL queries. The SPARQL semantics for compound patterns is defined by merging the result sets of BGPs. We proposed replacement semantics to solve the CSPs associated with the BGPs sequentially, taking into account the partial solution found so far. Such semantics is better suited for use in a tree search. We proved that the semantics are correct, under some conditions that are not restrictive in real-life queries.

Based on the denotational CSP semantics, we proposed an operational CP semantics that can be implemented with off-the-shelf CP solvers that allow posting constraints during the search. Experimental evaluation with the Comet [Dyn10] solver shows us that such approach is feasible, albeit not efficient (see chapter 7). The problem is that off-the-shelf solvers are not optimized for handling huge domains and tables.

In chapter 6, we introduced Castor embedding a specialized CP solver designed with large domains in mind. The key idea is to avoid as much as possible data structures and algorithms whose time or space complexity is proportional to the database size. The table constraints achieve forward-checking consistency with an on-disk table structure borrowed from RDF-3X [NW08].

We use a sparse set representation for finite domains. We complemented the representation with a bound representation that is synchronized lazily with the sparse set representation. Thanks to the dual representation, all operations on the domains are performed in constant time. We also extended the sparse set representation to allow for value-based propagation with a constraint-based queue.

Experimental evaluation (see chapter 7) shows the efficiency of our lightweight approach. Castor is competitive with state-of-the-art SPARQL engines, even on large databases. On complex queries with filters involving multiple variables, Castor is able to outperform state-of-the-art engines. However, we also observed that the performances of Castor on simple queries degrade quicker than state-of-the-art engines when increasing the database size. Our approach does not scale as well as relational database technology.

8.2 Perspectives

Castor is a research prototype and has several limitations as detailed in chapter 6. Apart from those implementation limitations, two research directions emerge for future work: increasing the performances and improving the expressiveness.

8.2.1 Performances

On the performance side, one open research question is how to deal with filters on compound patterns that cannot be pushed down onto basic graph patterns. If we exploit such filters during the search, solutions of an OPT or DIFF pattern could be pruned, yielding erroneous results. Hence, those filters are currently post-processed. However, one could investigate whether there are cases where some amount of pruning could be performed.

The search heuristics used by Castor are very basic. While the standard heuristics all seem more or less equally good (see chapter 7), we did not evaluate specialized heuristics. One could use the statistics generated by relational databases, such as selectivity estimates, to provide such specialized search heuristics. The statistics could also be used to order the propagators that are waiting to be called in the propagation queue. As we do not maintain domain consistency, the order in which the propagators are called may impact the resulting pruning.

A broader question is the potential combination of relational database technology (RDB) and constraint programming. RDB gives one point of view on how to solve SPARQL queries. This thesis provides an alternative point of view based on CP. Some features are considered an optimization in one domain, while they come standard in the other. For example, exploiting filters during the search is standard in CP, while it is an optimization in RDB.

By propagating each triple pattern separately, Castor basically performs a kind of nested loop join. There are much more efficient join operators in relational databases. To take advantage of such operators, one could group multiple triple patterns and propagate those together using RDB techniques. An open question is how many and which triple patterns should be grouped.

Similarly, we could use RDB to pre-process the query. In such case, RDB would process the query, joining result sets together, up to the point where the results of a join would grow too large. Then, CP would take over the search with the hope that filters would be able to reduce the search space.

8.2.2 Expressiveness

Castor currently implements most of SPARQL 1.0 features. Recently, the W3C has standardized an update to the language, SPARQL 1.1 [HS13]. One notable new feature is the ability to specify property paths in place of triples in basic graph patterns. Such property paths describe a path in the RDF graph with a language that is similar to regular expressions. Specialized propagators could be designed to handle such path constraints efficiently.

The SPARQL language allows for engine-specific extensions through custom functions in filter expressions. Using this facility, a large number of additional constraints could be provided to the user. However, the propagators of such constraints would have to be able to handle the large domains of the variables.

Throughout the thesis, we have ignored reasoning mechanisms, such as RDF-Schema and OWL. By applying rules, such mechanisms can infer additional triples in an RDF graph. To use Castor with the additional triples, we could compute the full deductive closure of the input graph. However, such approach is very inefficient. A better alternative would be to somehow integrate the inference rules in the CP model. We would need to redefine how a triple pattern is translated to constraints in the CSP corresponding to the basic graph pattern.

For example, if we consider an RDF Schema class hierarchy, a triple pattern $(p, \texttt{rdf:type}, \texttt{foaf:Agent})$ could be translated to the $\text{Member}((p, \texttt{rdf:type}, x), G)$ constraint, introducing an additional variable x. The domain of x would be `foaf:Agent` and all of its subclasses.

One could also be interested in the qualitative aspects of the solutions. SPARQL allows to order the solutions by some criteria, defined as expressions involving the query variables. We could enhance the ordering by taking into account other parameters, such as the trustworthiness or the relevance of the involved triples.

Finally, the SPARQL standard only defines complete evaluation of queries, i.e., we are looking for all the solutions. Doing so makes sense under a closed-world assumption, where an absent statement means the statement is false. The graph is thus assumed to contain all the knowledge. However, under an open-world assumption, we cannot guarantee that we dispose of the entire knowledge. The open-world assumption is common in the Semantic Web. In such a setting, finding all the solutions might be irrelevant, because there is no guarantee that there cannot be any more solutions. Instead, one might be interested in solving the query approximately. Random restarts or large neighborhood search can greatly increase the speed of the CP search, at the cost of losing completeness.

Appendices

Appendix A

Benchmark Queries

This appendix contains the complete SPARQL queries that were used in the benchmarks described in chapter 7. To improve clarity, the prefix definitions are omitted.

A.1 SPARQL Performance Benchmark

Here, we present the queries of the SPARQL Performance Benchmark [Sch+09]. The descriptions of the queries are those given by the authors of the benchmark. Where applicable, we also show the simplified queries that were used to compare with Comet (see section 7.3).

S1

Return the year of publication of "Journal 1 (1940)".

```
SELECT ?yr
WHERE {
  ?journal rdf:type bench:Journal .
  ?journal dc:title "Journal 1 (1940)"^^xsd:string .
  ?journal dcterms:issued ?yr
}
```

S2

Extract all inproceedings with properties `dc:creator`, `bench:booktitle`, `dc:title`, `swrc:pages`, `dcterms:partOf`, `rdfs:seeAlso`, `foaf:homepage`, `dcterms:issued`, and optionally `bench:abstract`, including these properties.

```
SELECT ?inproc ?author ?booktitle ?title
       ?proc ?ee ?page ?url ?yr ?abstract
```

```
WHERE {
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?author .
  ?inproc bench:booktitle ?booktitle .
  ?inproc dc:title ?title .
  ?inproc dcterms:partOf ?proc .
  ?inproc rdfs:seeAlso ?ee .
  ?inproc swrc:pages ?page .
  ?inproc foaf:homepage ?url .
  ?inproc dcterms:issued ?yr
  OPTIONAL {
    ?inproc bench:abstract ?abstract
  }
}
ORDER BY ?yr
```

Simplified query S2′:

```
SELECT ?inproc ?author ?booktitle ?title
       ?proc ?ee ?page ?url ?yr ?abstract
WHERE {
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?author .
  ?inproc bench:booktitle ?booktitle .
  ?inproc dc:title ?title .
  ?inproc dcterms:partOf ?proc .
  ?inproc rdfs:seeAlso ?ee .
  ?inproc swrc:pages ?page .
  ?inproc foaf:homepage ?url .
  ?inproc dcterms:issued ?yr
  OPTIONAL {
    ?inproc bench:abstract ?abstract
  }
}
```

S3a

Select all articles with property `swrc:pages`.

```
SELECT ?article
WHERE {
  ?article rdf:type bench:Article .
  ?article ?property ?value
  FILTER (?property = swrc:pages)
}
```

S3b

Select all articles with property `swrc:month`.

```
SELECT ?article
WHERE {
  ?article rdf:type bench:Article .
  ?article ?property ?value
  FILTER (?property = swrc:month)
}
```

S3c

Select all articles with property `swrc:isbn`.

```
SELECT ?article
WHERE {
  ?article rdf:type bench:Article .
  ?article ?property ?value
  FILTER (?property = swrc:isbn)
}
```

S4

Select all distinct pairs of article author names for authors that have published in the same journal.

```
SELECT DISTINCT ?name1 ?name2
WHERE {
  ?article1 rdf:type bench:Article .
  ?article2 rdf:type bench:Article .
  ?article1 dc:creator ?author1 .
  ?author1 foaf:name ?name1 .
  ?article2 dc:creator ?author2 .
  ?author2 foaf:name ?name2 .
  ?article1 swrc:journal ?journal .
  ?article2 swrc:journal ?journal
  FILTER (?name1 < ?name2)
}
```

Simplified query S4′:

```
SELECT ?name1 ?name2
WHERE {
  ?article1 rdf:type bench:Article .
  ?article2 rdf:type bench:Article .
```

```
    ?article1 dc:creator ?author1 .
    ?author1 foaf:name ?name1 .
    ?article2 dc:creator ?author2 .
    ?author2 foaf:name ?name2 .
    ?article1 swrc:journal ?journal .
    ?article2 swrc:journal ?journal
    FILTER (?name1 < ?name2)
  }
```

S5a

Return the names of all persons that occur as author of at least one inproceeding and at least one article.

```
  SELECT DISTINCT ?person ?name
  WHERE {
    ?article rdf:type bench:Article .
    ?article dc:creator ?person .
    ?inproc rdf:type bench:Inproceedings .
    ?inproc dc:creator ?person2 .
    ?person foaf:name ?name .
    ?person2 foaf:name ?name2
    FILTER (?name = ?name2)
  }
```

Simplified query S5a′:

```
  SELECT ?person ?name
  WHERE {
    ?article rdf:type bench:Article .
    ?article dc:creator ?person .
    ?inproc rdf:type bench:Inproceedings .
    ?inproc dc:creator ?person2 .
    ?person foaf:name ?name .
    ?person2 foaf:name ?name2
    FILTER (?name = ?name2)
  }
```

S5b

Return the names of all persons that occur as author of at least one inproceeding and at least one article (same as S5a).

```
  SELECT DISTINCT ?person ?name
  WHERE {
    ?article rdf:type bench:Article .
    ?article dc:creator ?person .
    ?inproc rdf:type bench:Inproceedings .
    ?inproc dc:creator ?person .
    ?person foaf:name ?name
  }
```

Simplified query S5b′:

```
  SELECT ?person ?name
  WHERE {
    ?article rdf:type bench:Article .
    ?article dc:creator ?person .
    ?inproc rdf:type bench:Inproceedings .
    ?inproc dc:creator ?person .
    ?person foaf:name ?name
  }
```

S6

Return, for each year, the set of all publications authored by persons that have not published in years before.

```
  SELECT ?yr ?name ?document
  WHERE {
    ?class rdfs:subClassOf foaf:Document .
    ?document rdf:type ?class .
    ?document dcterms:issued ?yr .
    ?document dc:creator ?author .
    ?author foaf:name ?name
    OPTIONAL {
      ?class2 rdfs:subClassOf foaf:Document .
      ?document2 rdf:type ?class2 .
      ?document2 dcterms:issued ?yr2 .
      ?document2 dc:creator ?author2
      FILTER (?author = ?author2 && ?yr2 < ?yr)
    } FILTER (!bound(?author2))
  }
```

S7

Return the titles of all papers that have been cited at least once, but not by any paper that has not been cited itself.

```
  SELECT DISTINCT ?title
```

```
WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?doc rdf:type ?class .
  ?doc dc:title ?title .
  ?bag2 ?member2 ?doc .
  ?doc2 dcterms:references ?bag2
  OPTIONAL {
    ?class3 rdfs:subClassOf foaf:Document .
    ?doc3 rdf:type ?class3 .
    ?doc3 dcterms:references ?bag3 .
    ?bag3 ?member3 ?doc
    OPTIONAL {
      ?class4 rdfs:subClassOf foaf:Document .
      ?doc4 rdf:type ?class4 .
      ?doc4 dcterms:references ?bag4 .
      ?bag4 ?member4 ?doc3
    } FILTER (!bound(?doc4))
  } FILTER (!bound(?doc3))
}
```

Simplified query S7′:

```
SELECT ?title
WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?doc rdf:type ?class .
  ?doc dc:title ?title .
  ?bag2 ?member2 ?doc .
  ?doc2 dcterms:references ?bag2
  OPTIONAL {
    ?class3 rdfs:subClassOf foaf:Document .
    ?doc3 rdf:type ?class3 .
    ?doc3 dcterms:references ?bag3 .
    ?bag3 ?member3 ?doc
    OPTIONAL {
      ?class4 rdfs:subClassOf foaf:Document .
      ?doc4 rdf:type ?class4 .
      ?doc4 dcterms:references ?bag4 .
      ?bag4 ?member4 ?doc3
    } FILTER (!bound(?doc4))
  } FILTER (!bound(?doc3))
}
```

S8

Compute authors that have published with “Paul Erdoes”, or with an author that has published with “Paul Erdoes”.

```
SELECT DISTINCT ?name
WHERE {
  ?erdoes rdf:type foaf:Person .
  ?erdoes foaf:name "Paul Erdoes"^^xsd:string .
  {
    ?document dc:creator ?erdoes .
    ?document dc:creator ?author .
    ?document2 dc:creator ?author .
    ?document2 dc:creator ?author2 .
    ?author2 foaf:name ?name
    FILTER (?author != ?erdoes &&
            ?document2 != ?document &&
            ?author2 != ?erdoes &&
            ?author2 != ?author)
  } UNION {
    ?document dc:creator ?erdoes.
    ?document dc:creator ?author.
    ?author foaf:name ?name
    FILTER (?author != ?erdoes)
  }
}
```

Simplified query S8′:

```
SELECT ?name
WHERE {
  ?erdoes rdf:type foaf:Person .
  ?erdoes foaf:name "Paul Erdoes"^^xsd:string .
  {
    ?document dc:creator ?erdoes .
    ?document dc:creator ?author .
    ?document2 dc:creator ?author .
    ?document2 dc:creator ?author2 .
    ?author2 foaf:name ?name
    FILTER (?author != ?erdoes &&
            ?document2 != ?document &&
            ?author2 != ?erdoes &&
            ?author2 != ?author)
  } UNION {
    ?document dc:creator ?erdoes.
```

```
      ?document dc:creator ?author.
      ?author foaf:name ?name
      FILTER (?author != ?erdoes)
    }
  }
```

S9

Return incoming and outcoming properties of persons.

```
  SELECT DISTINCT ?predicate
  WHERE {
    {
      ?person rdf:type foaf:Person .
      ?subject ?predicate ?person
    } UNION {
      ?person rdf:type foaf:Person .
      ?person ?predicate ?object
    }
  }
```

Simplified query S9′:

```
  SELECT ?predicate
  WHERE {
    {
      ?person rdf:type foaf:Person .
      ?subject ?predicate ?person
    } UNION {
      ?person rdf:type foaf:Person .
      ?person ?predicate ?object
    }
  }
```

S10

Return all subjects that stand in any relation to “Paul Erdoes”. In our scenario, the query might also be formulated as: return publications and venues in which “Paul Erdoes” is involved either as author or as editor.

```
  SELECT ?subject ?predicate
  WHERE {
    ?subject ?predicate person:Paul_Erdoes
  }
```

S11

Return (up to) 10 electronic edition URLs starting from the 51th publication, in lexicographical order.

```
SELECT ?ee
WHERE {
  ?publication rdfs:seeAlso ?ee
}
ORDER BY ?ee
LIMIT 10
OFFSET 50
```

A.2 Berlin SPARQL Benchmark

Here, we present the queries of the Berlin SPARQL Benchmark [BS09]. The descriptions of the queries are those given by the authors of the benchmark.

B1

Find products for a given set of generic features.

```
SELECT DISTINCT ?product ?label
WHERE {
  ?product rdfs:label ?label .
  ?product a %ProductType% .
  ?product bsbm:productFeature %ProductFeature1% .
  ?product bsbm:productFeature %ProductFeature2% .
  ?product bsbm:productPropertyNumeric1 ?value1 .
  FILTER (?value1 > %x%)
}
ORDER BY ?label
LIMIT 10
```

B2

Retrieve basic information about a specific product for display purposes.

```
SELECT ?label ?comment ?producer ?productFeature
       ?propertyTextual1 ?propertyTextual2 ?propertyTextual3
       ?propertyNumeric1 ?propertyNumeric2 ?propertyTextual4
       ?propertyTextual5 ?propertyNumeric4
WHERE {
  %ProductXYZ% rdfs:label ?label .
```

```
    %ProductXYZ% rdfs:comment ?comment .
    %ProductXYZ% bsbm:producer ?p .
    ?p rdfs:label ?producer .
    %ProductXYZ% dc:publisher ?p .
    %ProductXYZ% bsbm:productFeature ?f .
    ?f rdfs:label ?productFeature .
    %ProductXYZ% bsbm:productPropertyTextual1 ?propertyTextual1 .
    %ProductXYZ% bsbm:productPropertyTextual2 ?propertyTextual2 .
    %ProductXYZ% bsbm:productPropertyTextual3 ?propertyTextual3 .
    %ProductXYZ% bsbm:productPropertyNumeric1 ?propertyNumeric1 .
    %ProductXYZ% bsbm:productPropertyNumeric2 ?propertyNumeric2 .
    OPTIONAL {
      %ProductXYZ% bsbm:productPropertyTextual4 ?propertyTextual4
    }
    OPTIONAL {
      %ProductXYZ% bsbm:productPropertyTextual5 ?propertyTextual5
    }
    OPTIONAL {
      %ProductXYZ% bsbm:productPropertyNumeric4 ?propertyNumeric4
    }
  }
```

B3

Find products having some specific features and not having one feature.

```
  SELECT ?product ?label
  WHERE {
    ?product rdfs:label ?label .
    ?product a %ProductType% .
    ?product bsbm:productFeature %ProductFeature1% .
    ?product bsbm:productPropertyNumeric1 ?p1 .
    FILTER (?p1 > %x%)
    ?product bsbm:productPropertyNumeric3 ?p3 .
    FILTER (?p3 < %y%)
    OPTIONAL {
      ?product bsbm:productFeature %ProductFeature2% .
      ?product rdfs:label ?testVar
    }
    FILTER (!bound(?testVar))
  }
  ORDER BY ?label
  LIMIT 10
```

B4

Find products matching two different sets of features.

```
SELECT DISTINCT ?product ?label ?propertyTextual
WHERE {
  {
    ?product rdfs:label ?label .
    ?product rdf:type %ProductType% .
    ?product bsbm:productFeature %ProductFeature1% .
    ?product bsbm:productFeature %ProductFeature2% .
    ?product bsbm:productPropertyTextual1 ?propertyTextual .
    ?product bsbm:productPropertyNumeric1 ?p1 .
    FILTER (?p1 > %x%)
  } UNION {
    ?product rdfs:label ?label .
    ?product rdf:type %ProductType% .
    ?product bsbm:productFeature %ProductFeature1% .
    ?product bsbm:productFeature %ProductFeature3% .
    ?product bsbm:productPropertyTextual1 ?propertyTextual .
    ?product bsbm:productPropertyNumeric2 ?p2 .
    FILTER (?p2 > %y%)
  }
}
ORDER BY ?label
OFFSET 5
LIMIT 10
```

B5

Find product that are similar to a given product.

```
SELECT DISTINCT ?product ?productLabel
WHERE {
  ?product rdfs:label ?productLabel .
  FILTER (%ProductXYZ% != ?product)
  %ProductXYZ% bsbm:productFeature ?prodFeature .
  ?product bsbm:productFeature ?prodFeature .
  %ProductXYZ% bsbm:productPropertyNumeric1 ?origProperty1 .
  ?product bsbm:productPropertyNumeric1 ?simProperty1 .
  FILTER (?simProperty1 < (?origProperty1 + 120) &&
          ?simProperty1 > (?origProperty1 - 120))
  %ProductXYZ% bsbm:productPropertyNumeric2 ?origProperty2 .
  ?product bsbm:productPropertyNumeric2 ?simProperty2 .
  FILTER (?simProperty2 < (?origProperty2 + 170) &&
          ?simProperty2 > (?origProperty2 - 170))
}
ORDER BY ?productLabel
LIMIT 5
```

B7

Retrieve in-depth information about a specific product including offers and reviews.

```
SELECT ?productLabel ?offer ?price ?vendor ?vendorTitle
       ?review ?revTitle ?reviewer ?revName ?rating1 ?rating2
WHERE {
  %ProductXYZ% rdfs:label ?productLabel .
  OPTIONAL {
    ?offer bsbm:product %ProductXYZ% .
    ?offer bsbm:price ?price .
    ?offer bsbm:vendor ?vendor .
    ?vendor rdfs:label ?vendorTitle .
    ?vendor bsbm:country
               <http://downlode.org/rdf/iso-3166/countries#DE> .
    ?offer dc:publisher ?vendor .
    ?offer bsbm:validTo ?date .
    FILTER (?date > %currentDate%)
  }
  OPTIONAL {
    ?review bsbm:reviewFor %ProductXYZ% .
    ?review rev:reviewer ?reviewer .
    ?reviewer foaf:name ?revName .
    ?review dc:title ?revTitle .
    OPTIONAL { ?review bsbm:rating1 ?rating1 . }
    OPTIONAL { ?review bsbm:rating2 ?rating2 . }
  }
}
```

B8

Give me recent reviews in English for a specific product.

```
SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName
       ?rating1 ?rating2 ?rating3 ?rating4
WHERE {
  ?review bsbm:reviewFor %ProductXYZ% .
  ?review dc:title ?title .
  ?review rev:text ?text .
```

```
    FILTER (langMatches(lang(?text), "EN"))
    ?review bsbm:reviewDate ?reviewDate .
    ?review rev:reviewer ?reviewer .
    ?reviewer foaf:name ?reviewerName .
    OPTIONAL { ?review bsbm:rating1 ?rating1 . }
    OPTIONAL { ?review bsbm:rating2 ?rating2 . }
    OPTIONAL { ?review bsbm:rating3 ?rating3 . }
    OPTIONAL { ?review bsbm:rating4 ?rating4 . }
  }
  ORDER BY DESC(?reviewDate)
  LIMIT 20
```

B11

Get all information about an offer.

```
  SELECT ?property ?hasValue ?isValueOf
  WHERE {
    { %OfferXYZ% ?property ?hasValue }
    UNION
    { ?isValueOf ?property %OfferXYZ% }
  }
```

Bibliography

[Aba+09] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. “SW-Store: a vertically partitioned DBMS for Semantic Web data management”. In: *The VLDB Journal* 18 (2 Apr. 2009), pp. 385–406. ISSN: 1066-8888.

[ACM10] Marcelo Arenas, Mariano Consens, and Alejandro Mallea. “Revisiting Blank Nodes in RDF to Avoid the Semantic Mismatch with SPARQL”. In: *W3C Workshop: RDF Next Steps*. 2010.

[AG08] Renzo Angles and Claudio Gutierrez. “The Expressive Power of SPARQL”. In: *The Semantic Web – ISWC 2008*. Ed. by Amit Sheth et al. Vol. 5318. Lecture Notes in Computer Science. Springer, 2008, pp. 114–129.

[Bag05] Jean-François Baget. “RDF Entailment as a Graph Homomorphism”. In: *The Semantic Web – ISWC 2005*. Vol. 3729. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 82–96. ISBN: 978-3-540-29754-3.

[Bec01] David Beckett. “The Design and Implementation of the Redland RDF Application Framework”. In: *10th International World Wide Web Conference*. 2001.

[Bel+08] François Belleau et al. “Bio2RDF: Towards a mashup to build bioinformatics knowledge systems”. In: *Journal of Biomedical Informatics* 41.5 (2008), pp. 706–716. ISSN: 1532-0464.

[Bes+99] Christian Bessière, Pedro Meseguer, EugeneC. Freuder, and Javier Larrosa. “On Forward Checking for Non-binary Constraint Satisfaction”. In: *Principles and Practice of Constraint Programming (CP’99)*. Ed. by Joxan Jaffar. Vol. 1713. Lecture Notes in Computer Science. Springer, 1999, pp. 88–102. ISBN: 978-3-540-66626-4.

[Bes06] Christian Bessiere. *Handbook of Constraint Programming, chapter 3*. Elsevier Science Inc., 2006.

[BFM05] T. Berners-Lee, R. Fielding, and L. Masinter. *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax*. Jan. 2005. URL: `http://www.ietf.org/rfc/rfc3986.txt`.

[BHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. “The semantic web”. In: *Scientific American* 284.5 (2001), pp. 28–37.

[BK11] Robert Battle and Dave Kolas. “Linking Geospatial Data With GeoSPARQL”. In: *Semantic Web – Interoperability, Usability, Applications* 24 (2011).

[BKH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. “Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema”. In: *Proceedings of the First International Semantic Web Conference on The Semantic Web*. ISWC 2002. Springer, 2002, pp. 54–68. ISBN: 3-540-43760-6.

[BM10] Dan Brickley and Libby Miller. *FOAF Vocabulary Specification 0.98*. 2010. URL: http://xmlns.com/foaf/spec/.

[Bra07] Steve Bratt. *Semantic Web, and Other Technologies to Watch*. Slides presented at the 2007 INCOSE International Workshop. Jan. 2007.

[BS09] Christian Bizer and Andreas Schultz. “The Berlin SPARQL Benchmark”. In: *International Journal on Semantic Web and Information Systems (IJSWIS)* 5.2 (2009), pp. 1–24.

[BT93] P. Briggs and L. Torczon. “An efficient representation for sparse sets”. In: *ACM Letters on Programming Languages and Systems* 2.1–4 (1993), pp. 59–69.

[Car+04] Jeremy J. Carroll et al. “Jena: implementing the semantic web recommendations”. In: *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*. WWW Alt. ’04. New York, NY, USA: ACM, 2004, pp. 74–83. ISBN: 1-58113-912-8.

[CDS09] V. le Clément, Y. Deville, and C. Solnon. “Constraint-based Graph Matching”. In: *15th Conference on Principles and Practice of Constraint Programming (CP)*. Vol. 5732. LNCS. Springer, 2009, pp. 274–288.

[CDS11] Vianney le Clément de Saint-Marcq, Yves Deville, and Christine Solnon. “An Efficient Light Solver for Querying the Semantic Web”. In: *Principles and Practice of Constraint Programming – CP 2011*. Ed. by Jimmy Lee. Vol. 6876. Lecture Notes in Computer Science. Springer, 2011, pp. 145–159. ISBN: 978-3-642-23785-0.

[CFT08] Kendall Grant Clark, Lee Feigenbaum, and Elias Torres. *SPARQL Protocol for RDF*. W3C. Jan. 2008. URL: http://www.w3.org/TR/rdf-sparql-protocol/.

[CHM11] Michael J. Cafarella, Alon Halevy, and Jayant Madhavan. “Structured data on the web”. In: *Commun. ACM* 54 (2 Feb. 2011), pp. 72–79. ISSN: 0001-0782.

[Clé+12] Vianney le Clément de Saint-Marcq, Yves Deville, Christine Solnon, and Pierre-Antoine Champin. “Castor: A Constraint-Based SPARQL Engine with Active Filter Processing”. In: *The Semantic Web: Research and Applications*. Ed. by Elena Simperl et al. Vol. 7295. Lecture Notes in Computer Science. Springer, 2012, pp. 391–405. ISBN: 978-3-642-30283-1.

[Clé+13] Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. “Sparse-Sets for Domain Implementation”. In: *Techniques for implementing constraint programming systems (TRICS) workshop at CP 2013*. 2013.

[Com79] Douglas Comer. “The Ubiquitous B-Tree”. In: *ACM Comput. Surv.* 11.2 (June 1979), pp. 121–137. ISSN: 0360-0300.

[CY10] KenilC.K. Cheng and RolandH.C. Yap. “An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints”. English. In: *Constraints* 15.2 (2010), pp. 265–304. ISSN: 1383-7133.

[DS04] Mike Dean and Guus Schreiber. *OWL Web Ontology Language*. W3C. Feb. 2004. URL: `http://www.w3.org/TR/owl-ref/`.

[DS05] M. Duerst and M. Suignard. *RFC 3987: Internationalized Resource Identifiers (IRIs)*. Jan. 2005. URL: `http://www.ietf.org/rfc/rfc3987.txt`.

[Dyn10] Dynamic Decision Technologies Inc. *Comet*. 2010. URL: `http://www.dynadec.com`.

[EM09] Orri Erling and Ivan Mikhailov. “RDF Support in the Virtuoso DBMS”. In: *Networked Knowledge – Networked Media*. Vol. 221. Studies in Computational Intelligence. Springer, 2009, pp. 7–24.

[Erl12] Orri Erling. “Virtuoso, a hybrid RDBMS/graph column store”. In: *IEEE Data Eng. Bull* 35.1 (2012), pp. 3–8.

[Hay04] Patrick Hayes. *RDF Semantics*. W3C. Feb. 2004. URL: `http://www.w3.org/TR/2004/REC-rdf-mt-20040210/`.

[Hém+08] Sándor Héman, Niels Nes, Marcin Zukowski, and Peter Boncz. “Positional Delta Trees to reconcile updates with read-optimized data storage”. In: *CWI. Information Systems [INS]* E0801 (2008), pp. 1–11.

[HLS09] Steve Harris, Nick Lamb, and Nigel Shadbolt. “4store: The Design and Implementation of a Clustered RDF Store”. In: *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2009), at ISWC 2009*. 2009.

[Hor05] Herman J. ter Horst. “Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary”. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 3.2–3 (2005), pp. 79–115. ISSN: 1570-8268.

[Hos+11] Katja Hose, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. “Database Foundations for Scalable RDF Processing”. In: *Reasoning Web. Semantic Technologies for the Web of Data*. Vol. 6848. Lecture Notes in Computer Science. Springer, 2011, pp. 202–249. ISBN: 978-3-642-23031-8.

[HS13] Steve Harris and Andy Seaborne. *SPARQL 1.1 Query Language*. W3C. Mar. 2013. URL: `http://www.w3.org/TR/2013/REC-sparql11-query-20130321/`.

[Idr+12] Stratos Idreos et al. “MonetDB: Two Decades of Research in Column-oriented Database Architectures”. In: *Data Engineering* (2012), p. 40.

[KCM04] G. Klyne, J. J. Carroll, and B. McBride. *Resource description framework (RDF): Concepts and abstract syntax*. W3C. Feb. 2004. URL: `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`.

[Lec09] Christophe Lecoutre. *Constraint Networks: Techniques and Algorithms*. ISTE, 2009.

[Lec11] Christophe Lecoutre. “STR2: optimized simple tabular reduction for table constraints”. English. In: *Constraints* 16.4 (2011), pp. 341–371. ISSN: 1383-7133.

[Leh+13] Jens Lehmann et al. “DBpedia – A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia”. In: *Semantic Web Journal* (2013). Under review.

[Ley13] Michael Ley. *DBLP Database*. 2013. URL: `http://www.informatik.uni-trier.de/~ley/db/`.

[LM09] J.J. Levandoski and M.F. Mokbel. “RDF Data-Centric Storage”. In: *Web Services, 2009. ICWS 2009. IEEE International Conference on*. 2009, pp. 911–918.

[Luo+12] Yongming Luo et al. “Storing and indexing massive RDF datasets”. In: *Semantic Search over the Web*. Springer, 2012, pp. 31–60.

[LV02] J. Larrosa and G. Valiente. “Constraint satisfaction algorithms for graph pattern matching”. In: *Mathematical. Structures in Comp. Sci.* 12.4 (2002), pp. 403–422.

[Mal+11] Alejandro Mallea, Marcelo Arenas, Aidan Hogan, and Axel Polleres. “On Blank Nodes”. In: *The Semantic Web – ISWC 2011*. Vol. 7031. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 421–437. ISBN: 978-3-642-25072-9.

[Mor68] Donald R. Morrison. “PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric”. In: *J. ACM* 15.4 (Oct. 1968), pp. 514–534. ISSN: 0004-5411.

[NW08] Thomas Neumann and Gerhard Weikum. “RDF-3X: a RISC-style engine for RDF”. In: *Proc. VLDB Endow.* 1 (1 Aug. 2008), pp. 647–659. ISSN: 2150-8097.

[NW09] Thomas Neumann and Gerhard Weikum. “Scalable join processing on very large RDF graphs”. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. SIGMOD ’09. Providence, Rhode Island, USA: ACM, 2009, pp. 627–640. ISBN: 978-1-60558-551-2.

[NW10] Thomas Neumann and Gerhard Weikum. “x-RDF-3X: fast querying, high update rates, and consistency for RDF databases”. In: *Proc. VLDB Endow.* 3.1–2 (2010), pp. 256–263. ISSN: 2150-8097.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. “Semantics and complexity of SPARQL”. In: *ACM Trans. Database Syst.* 34 (3 Sept. 2009), 16:1–16:45. ISSN: 0362-5915.

[PS08] Eric Prud’hommeaux and Andy Seaborne. *SPARQL Query Language for RDF*. W3C. Jan. 2008. URL: `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`.

[SA10] Sherif Sakr and Ghazi Al-Naymat. “Relational processing of RDF queries: a survey”. In: *SIGMOD Rec.* 38.4 (June 2010), pp. 23–28. ISSN: 0163-5808.

[SC06] Christian Schulte and Mats Carlsson. “Finite domain constraint programming systems”. In: *Handbook of Constraint Programming* (2006), pp. 495–526.

[Sce+07] Simon Scerri, Michael Sintek, Ludger van Elst, and Siegfried Handschuh. *NEPOMUK Annotation Ontology Specification*. 2007. URL: `http://www.semanticdesktop.org/ontologies/nao/`.

[Sch+09] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. “SP^{2}Bench: A SPARQL Performance Benchmark”. In: *Proc. IEEE 25th Int. Conf. Data Engineering ICDE ’09*. 2009, pp. 222–233.

[SML10] Michael Schmidt, Michael Meier, and Georg Lausen. “Foundations of SPARQL query optimization”. In: *Proceedings of the 13th International Conference on Database Theory*. ICDT ’10. Lausanne, Switzerland: ACM, 2010, pp. 4–33. ISBN: 978-1-60558-947-3.

[Sol10] Christine Solnon. “AllDifferent-based Filtering for Subgraph Isomorphism”. In: *Artificial Intelligence* 174.12–13 (2010), pp. 850–864. ISSN: 0004-3702.

[Wei+98] Stuart Weibel, John Kunze, Carl Lagoze, and Misha Wolf. “Dublin core metadata for resource discovery”. In: *Internet Engineering Task Force RFC* 2413 (1998), p. 222.

[WKB08] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. “Hexastore: sextuple indexing for semantic web data management”. In: *Proc. VLDB Endow.* 1 (1 Aug. 2008), pp. 1008–1019. ISSN: 2150-8097.

[ZDS10] Stéphane Zampelli, Yves Deville, and Christine Solnon. “Solving subgraph isomorphism problems with constraint programming”. In: *Constraints* 15 (3 2010), pp. 327–353. ISSN: 1383-7133.