Chapter 1
On the Semantics of SPARQL

Marcelo Arenas, Claudio Gutierrez, Jorge Pérez

Abstract The Resource Description Framework (RDF) is the standard data model for representing information about World Wide Web resources. Jointly with its release as Recommendation of the W3C, the natural problem of querying RDF data was raised. In the last years, the language SPARQL has become the standard query language for RDF and, in fact, a W3C Recommendation since January 2008. In this chapter, we give a detailed description of the semantics of SPARQL. We start by focusing on the definition of a formal semantics for the core part of SPARQL, and then move to the definition for the entire language, including all the features in the specification of SPARQL by the W3C such as blank nodes in graph patterns and bag semantics for solutions.

1.1 Introduction

The Resource Description Framework (RDF) is a data model for representing information about World Wide Web resources. Jointly with its release in 1998 as Recommendation of the W3C, the natural problem of querying RDF data was raised. Since then, several designs and implementations of RDF query languages have been proposed. In 2004, the RDF Data Access Working Group, part of the W3C Semantic Web Activity, released a first public working draft of a query language for RDF,

Marcelo Arenas
Pontificia Universidad Católica de Chile, Department of Computer Science, Vicuña Mackenna 4860, 7820436 Macul, Santiago, Chile. e-mail: marenas@ing.puc.cl

Claudio Gutierrez
Universidad de Chile, Department of Computer Science, Blanco Encalada 2120, 8370459 Santiago, Santiago, Chile. e-mail: cgutierr@dcc.uchile.cl

Jorge Pérez
Pontificia Universidad Católica de Chile, Department of Computer Science, Vicuña Mackenna 4860, 7820436 Macul, Santiago, Chile. e-mail: jperez@ing.puc.cl

called SPARQL [15].[1] Since then, SPARQL has been rapidly adopted as the standard for querying Semantic Web data. In January 2008, SPARQL became a W3C Recommendation.

RDF is a directed labeled graph data format and, thus, SPARQL is essentially a graph-matching query language. SPARQL queries are composed by three parts. The *pattern matching part*, which includes several interesting features of pattern matching of graphs, like optional parts, union of patterns, nesting, filtering values of possible matchings, and the possibility of choosing the data source to be matched by a pattern. The *solution modifiers*, which once the output of the pattern has been computed (in the form of a table of values of variables), allow to modify these values applying classical operators like projection, distinct, order and limit. Finally, the *output* of a SPARQL query can be of different types: yes/no queries, selections of values of the variables which match the patterns, construction of new RDF data from these values, and descriptions of resources.

The definition of a formal semantics for SPARQL has played a key role in the standardization process of this query language. Although taken one by one the features of SPARQL are intuitive and simple to describe and understand, it turns out that the combination of them makes SPARQL into a complex language. Reaching a consensus in the W3C standardization process about a formal semantics for SPARQL was not an easy task. The initial efforts to define SPARQL were driven by use cases, mostly by specifying the expected output for particular example queries. In fact, the interpretations of examples and the exact outcomes of cases not covered in the initial drafts of the SPARQL specification, were a matter of long discussions in the W3C mailing lists. In [11], the authors presented one of the first formalizations of a semantics for a fragment of the language. Currently, the official specification of SPARQL [15], endorsed by the W3C, formalizes a semantics based on [11].

A formalization of a semantics for SPARQL is beneficial for several reasons, including to serve as a tool to identify and derive relations among the constructors that stay hidden in the use cases, to identify redundant and contradicting notions, to drive and help the implementation of query engines, and to study the complexity, expressiveness, and further natural database questions like rewriting and optimization. In this chapter, we present a streamlined version of the core fragment of SPARQL with precise algebraic syntax and a formal compositional semantics based on [11, 12, 13].

One of the delicate issues in the definition of a semantics for SPARQL is the treatment of *optional matching* and incomplete answers. The idea behind optional matching is to allow information to be added if the information is available in the data source, instead of just failing to give an answer whenever some part of the pattern does not match. This feature of optional matching is crucial in Semantic Web applications, and more specifically in RDF data management, where it is assumed that every application have only partial knowledge about the resources being managed. The semantics of SPARQL is formalized by using *partial mappings* between variables in the patterns and actual values in the RDF graph being queried. This formalization allows one to deal with partial answers in a clean way, and is based

[1] The name SPARQL is a recursive acronym that stands for *SPARQL Protocol and RDF Query Language*.

on the extension of some classical relational algebra operators to work over sets of partial mappings.

The rest of the chapter is organized as follows. In Section 1.2 we describe the official syntax of SPARQL proposed by the W3C. In Section 1.3 we introduce an algebraic syntax for the language and compare it with the official syntax. In Section 1.4 we formalize the semantics of SPARQL. We begin formalizing a set semantics for the language without considering blank nodes in patterns. We then extend the semantics to consider blank nodes and we provide a bag semantics for SPARQL. In Section 1.5 we review some of the results in the literature about the complexity of evaluating SPARQL graph patterns. Section 1.6 describes some related work about the formalization of a semantics for SPARQL. Concluding remarks are in Section 1.7.

1.2 The W3C Syntax of SPARQL

The RDF query language SPARQL was adopted as a W3C Recommendation on January 15, 2008. Its syntax and semantics is specified in [15]. SPARQL is a language designed to query data in the form of sets of triples, namely RDF graphs (see Section 1.3 for a formal definition of the notion of RDF graph). The basic engine of the language is a pattern matching facility, which uses some graph pattern matching functionalities (sets of triples can be viewed also as graphs). The overall structure of the language –from a syntactic point of view– resembles SQL with its three main blocks (as shown in Fig. 1.1):

- A WHERE clause, which is composed of a graph pattern. Informally speaking, this clause is given by a pattern that corresponds to an RDF graph where some resources have been replaced by variables. But not only that, more complex expressions (patterns) are also allowed, which are formed by using some algebraic operators. This pattern is used as a filter of the values of the dataset to be returned.
- A FROM clause, which specifies the sources or datasets to be queried.
- A SELECT clause, which specifies the final form in which the results are returned to the user. SPARQL, in contrast to SQL, allows several forms of returning the data: a table using SELECT, a graph using DESCRIBE or CONSTRUCT, or a TRUE/FALSE answer using ASK.

In what follows, we explain in more detail each component of the language. Of course, for ultimate details the reader should consult [15].

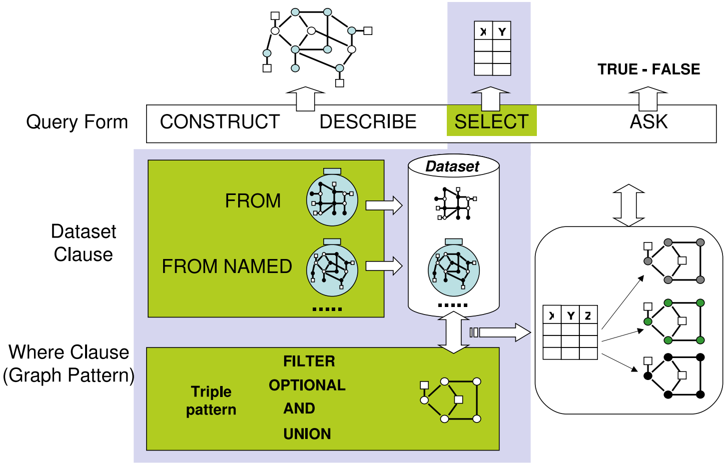

Fig. 1.1 The general form of a SPARQL query.

1.2.1 Basic definitions

There are several basic concepts used in the definition of the syntax of SPARQL, many of which are taken from the RDF specification with some minor modifications. For the sake of completeness, we review them here.

An *IRI* (Internationalized Resource Identifier [5]) is an identifier of resources, which essentially extends the syntax of URIs to a much wider repertoire of characters for internationalization purposes. For denoting resources, SPARQL uses IRIs instead of the URIs of RDF. A *literal* is used to identify values such as numbers and dates by means of a lexical representation. Anything represented by a literal could also be represented by a IRI, but it is often more convenient or intuitive to use literals. All literals have a lexical form that is a Unicode string. There are two types of literals: plain and typed. A *plain literal* is a string combined with an optional language tag. This may be used for plain text in a natural language. A *typed literal* is a string combined with a datatype IRI.

1.2.2 Basic structures

In order to present the language, we follow the grammar given in Fig. 1.2 that specifies the basic structure of the SPARQL Query Grammar [15]. [2]

```
Query       ::=  Prologue ( SelectQuery | ConstructQuery |
                            DescribeQuery | AskQuery )

SelectQuery ::=  "SELECT" ( "DISTINCT" | "REDUCED" )?
                          ( Var+ | "*" )
                 DatasetClause* WhereClause SolutionModifier

ConstructQuery ::=  "CONSTRUCT" ConstructTemplate
                    DatasetClause* WhereClause
                    SolutionModifier

DescribeQuery  ::=  "DESCRIBE" ( VarOrIRIref+ | "*" )
                    DatasetClause* WhereClause?
                    SolutionModifier

AskQuery       ::=  "ASK" DatasetClause* WhereClause

DatasetClause  ::=  "FROM" ( DefaultGraphClause |
                             NamedGraphClause )

WhereClause    ::=  "WHERE"? GroupGP

GroupGP        ::= "{" TB? ((GPNotTr | Filter) "."? TB?)* "}"
GPNotTr        ::= OptionalGP | GroupOrUnionGP | GraphGP
OptionalGP     ::= "OPTIONAL" GroupGP
GraphGP        ::= "GRAPH" VarOrIRIref GroupGP
GroupOrUnionGP ::= GroupGP ( "UNION" GroupGP )*
Filter         ::= "FILTER" Constraint

SolutionModifier  ::=  OrderClause? LimitOffsetClauses?
```

Fig. 1.2 A fragment of the SPARQL Query Grammar [15].

As shown in Fig. 1.2, a SPARQL `Query` is given by a `Prologue` followed by any of the four types of SPARQL queries: `SelectQuery`, `ConstructQuery`, `DescribeQuery` or `AskQuery`. The `Prologue` contains the declaration of variables, namespaces, and abbreviations to be used in the query. The `SELECT` clause in a `SelectQuery` selects a group of variables, or all of them using –as in SQL– the wildcard `*`. In this type of queries, one can eliminate duplicate solutions using `DISTINCT`. In a `ConstructQuery`, the `CONSTRUCT` form, and more specifically the `ConstructTemplate` form, is used to constructs an RDF graph using the obtained solutions. In a `DescribeQuery`, the `DESCRIBE` form

[2] http://www.w3.org/TR/rdf-sparql-query/#grammar

is not normative (only informative). It is intended to describe the specified variables or IRIs, i.e., it returns all the triples in the dataset involving these resources. In an `AskQuery`, the `ASK` form has no parameters but the dataset to be queried and a `WHERE` clause. It returns TRUE if the solution set is not empty, and FALSE otherwise.

In a SPARQL query, the `DatasetClause` allows to specify one graph (the `DefaultGraphClause`) or a set of named graphs, i.e., a set of pairs of identifiers and graphs, which are the data sources to be used when computing the answer to the query. Moreover, the `WHERE` clause is used to indicate how the information from the data sources is to be filtered, and it can be considered the central component of the query language. It specifies the pattern to be matched against the data sources. In particular, it includes sets of triples with some of the IRIs or blank elements replaced by variables, called "triple blocks" (`TB` in the grammar), an operator for collecting triples and blocks (denoted by `{A . B}`, and with no fixed arity), an operator `UNION` for specifying alternatives, an operator `OPTIONAL` to provide optional matchings, and an operator `FILTER` that allows filtering results of patterns under certain basic constraints.

Example 1.1. Consider the following query: "Give the name and the mailbox of each person who has a mailbox with domain .cl". This query can be expressed in SPARQL as follows:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
PREFIX ex:     <http://example.com/ns#>

SELECT ?name ?mbox
FROM  <myDataSource.rdf>
WHERE  {
         ?x  foaf:name  ?name .
         ?x  foaf:mbox  ?mbox .
         ?mbox  ex:domain  ".cl"
       }
```

The first two lines in this example form the `Prologue` of the query, which specifies the namespaces to be used. In this case, one is the well-known FOAF ontology, and the other one is an example namespace. The keywords `foaf` and `ex` are abbreviations for the namespaces, which are used in the body of the query.

The `SELECT` keyword indicates that the query returns a table with two columns, corresponding to the values obtained from the matching of the variables `?name` and `?mbox` against the graph pointed to in the `FROM` clause (`myDataSource.rdf`), and according to the pattern described in the `WHERE` clause. It should be noticed that a string starting with the symbol `?` denotes a variables in SPARQL.

In the above query, the `WHERE` clause is composed by a pattern with three triples: `?x foaf:name ?name`, `?x foaf:mbox ?mbox` and `?mbox ex:domain ".cl"`, where `.cl` is a literal. This pattern indicates that one is looking for the elements `?x`, `?name` and `?mbox` in the RDF graph `myDataSource.rdf` such that the `foaf:name` of `?x` is `?name`, the `foaf:mbox` of `?x` is `?mbox` and the `ex:domain` of `?mbox` is `.cl`. Thus, an expression of the form `{A . B}` in

SPARQL denotes the conjunction of A and B, as this expression holds if both A and B holds. □

1.2.3 More complex queries

SPARQL allows to write more complex queries than the ones presented in the previous section. The syntax of these queries becomes slightly involved and, in particular, two important issues are the use of the OPTIONAL and of the FILTER operator. We discuss these issues in this section.

Example 1.2 (Querying optional values). Consider the following query: "Give the name and the mailbox, if it is provided, of each person in the FOAF file of Bob". This query can be expressed in SPARQL as follows:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name ?mbox
FROM  <http://example.org/foaf/bobFoaf>
WHERE  {
         ?x  foaf:name ?name .
         OPTIONAL { ?x  foaf:mbox  ?mbox }
       }
```

In this case, the WHERE clause is composed by the conjunction of two patterns, the triple pattern ?x foaf:name ?name and the optional pattern:

```
OPTIONAL { ?x  foaf:mbox  ?mbox },
```

which in turn includes the triple pattern ?x foaf:mbox ?mbox. The WHERE clause indicates that one is looking for the elements ?name and ?mbox in the RDF graph http://example.org/foaf/bobFoaf such that the foaf:name of ?x is ?name and the foaf:mbox of ?x is ?mbox, if ?x has a foaf:mbox. In the case where ?x does not have a mailbox, the variable ?mbox is not instantiated and, thus, the corresponding tuple in the answer table only has a value in the attribute ?name. □

As shown in the previous example, the keyword OPTIONAL in the W3C SPARQL syntax works as a unary operator. In the following example, we show a query where this operator has to be used in conjunction with the notion of named graph.

Example 1.3 (Querying different data sources). Consider the following query: "For every person known by Alice and Bob, give the nicknames by which are known by Alice and Bob". We note that in this case the query must be posed against two different data sets, the FOAF data of Bob and of Alice. Moreover, it could be the case that Bob and Alice know different nicknames for a person, or that Bob knows a nickname for a person which is not known by Alice, or vice verse. Hence, to express the query in SPARQL, we need to use named graphs and the OPTIONAL operator:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX data: <http://example.org/foaf/>

SELECT ?nickA ?nickB
FROM NAMED <http://example.org/foaf/aliceFoaf>
FROM NAMED <http://example.org/foaf/bobFoaf>
WHERE
  {
    GRAPH data:bobFoaf { ?x foaf:knows ?comm .
                OPTIONAL { ?comm foaf:nick ?nickB } } .
    GRAPH data:aliceFoaf { ?y foaf:knows ?comm .
                OPTIONAL { ?comm foaf:nick ?nickA } }
  }
```

Notice that in the `WHERE` clause, the operator `GRAPH` is used to specify over which dataset the pattern enclosed in braces should be matched. Also, notice the use of the `OPTIONAL` operator to avoid loosing information for people that only has a registered nickname in the FOAF data of either Alice of Bob. □

It is important to notice that nesting of optional patterns is allowed in the official specification of SPARQL [15]. Unfortunately, the rules that define this nesting are rather involved (see rules 20-23 in [15]).

As mentioned above, the operator `FILTER` is another interesting and complex feature of SPARQL. More specifically, SPARQL filters restrict the solutions of a graph pattern match according to a given expression, which includes several functions and operators that are defined over the elements of the RDF graphs and the variables of SPARQL queries. A subset of these functions and operators are taken from XQuery and XPath (see [15] for further details). Among them, one of the most useful is the unary operator `bound(?x)`, which checks if the variable `?x` is bounded in the answer (this turns out to be really useful in combination with the `OPTIONAL` operator [1]). The functions `isIRI`, `isBlank` and `isLiteral` play similar roles. As expected, the Boolean connectives OR, AND and NOT (denoted by `logical-or`, `logical-and` and `!`, respectively) have also been included, as well as some functionalities for checking equality and order. The following example shows one of these features.

Example 1.4 (Filtering values). Consider the following query: "Give the list of people for whom Alice knows at least two nicknames". This query can be expressed by the following SPARQL query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?y
FROM <http://example.org/foaf/aliceFoaf>
WHERE
  {
    ?x foaf:knows ?y .
    ?y foaf:nick ?nick1 .
    ?y foaf:nick ?nick2 .
    FILTER (?nick1 != ?nick2) }
  }
```

The filter expression `FILTER (?nick1 != ?nick2)` is used to check that the nicknames `?nick1` and `?nick2` of `?y` are distinct. Thus, this expression is used to ensure that Alice knows at least two distinct nicknames for `?y`. □

We conclude this section by pointing out that one important aspect of SPARQL is the scope of the `FILTER` operator, which is a source of difficulties in the current specification of SPARQL (see [1]).

1.2.4 Final remarks

We conclude this section by providing a list of some important syntactic features of SPARQL, which are widely used in practice (for a complete list see the official specification of SPARQL [15]).

- A literal in SPARQL is a string (enclosed in either double quotes or single quotes), with either an optional language tag (introduced by `@`) or an optional datatype IRI or prefixed name (introduced by `^^`).
- Variables are prefixed by either `"?"` or `"$"`, and these two symbols are not considered to be part of the variable name. Furthermore, variables in SPARQL queries have global scope; the use of a given variable name anywhere in a query identifies the same variable.
- There is syntactic sugar for expressing namespaces. As we pointed out, in general it is more convenient to declare them in the `Prologue` of the query, but this is not mandatory. For example, these three expressions represent the same query:

```
PREFIX  dc: <http://purl.org/dc/elements/1.1/>
SELECT  ?title
WHERE   { <http://example.org/book/book1> dc:title ?title }

PREFIX  dc: <http://purl.org/dc/elements/1.1/>
PREFIX  : <http://example.org/book/>
SELECT  $title
WHERE   { :book1  dc:title  $title }

BASE    <http://example.org/book/>
PREFIX  dc: <http://purl.org/dc/elements/1.1/>
SELECT  $title
WHERE   { <book1>  dc:title  ?title }
```

  The clause `BASE` is used to indicate the base IRI for a query. In the last example, this base IRI is `http://example.org/book/` and, thus, the element `book1` in the query refers to this namespace.
- SPARQL allows a simplified notation for sets of triple patterns with a common subject; the symbol `;` can be used to express that a set of pairs is associated with a particular subject, thus writing the subject only one. For example, the following sequence of triples:

```
    ?x  foaf:name  ?name .
    ?x  foaf:mbox  ?mbox .
```

is the same in SPARQL as:

```
    ?x  foaf:name  ?name ;
        foaf:mbox  ?mbox .
```

1.3 An Algebraic Syntax for SPARQL

In this section, we present the algebraic formalization of the core fragment of SPARQL proposed in [11, 12, 13], and show that it is equivalent in expressive power to the core fragment of SPARQL. Thus, this formalization is used in this chapter to give a formal semantics to SPARQL, as well as to study some fundamental properties of this language.

We start by introducing the necessary notions about RDF (for details on the formalization of RDF see [7]). Assume that there are pairwise disjoint infinite sets I, B, and L (IRIs [5], Blank nodes, and Literals, respectively). A triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an *RDF triple*. In this tuple, s is the *subject*, p the *predicate* and o the *object*. We denote the union $I \cup B \cup L$ by T (RDF Terms). Assume additionally the existence of an infinite set V of variables disjoint from the above sets.

Definition 1.1 (RDF Graph). An *RDF graph* [10] is a set of RDF triples. If G is an RDF graph, then $\text{term}(G)$ is the set of elements of T appearing in the triples of G, and $\text{blank}(G)$ is the set of blank nodes appearing in G ($\text{blank}(G) = \text{term}(G) \cap B$).□

SPARQL queries are evaluated against an *RDF dataset* [15], that is, a set of RDF graphs in which every graph is identified by an IRI, except for a distinguished graph in the set called the *default* graph. Formally, an RDF dataset is a set:

$$\mathscr{D} = \{G_0, \langle u_1, G_1 \rangle, \ldots, \langle u_n, G_n \rangle\}$$

where $G_0, \ldots, G_n$ are RDF graphs, $u_1, \ldots, u_n$ are distinct IRIs, and $n \geq 0$. In the dataset, G_0 is the *default graph*, and the pairs $\langle u_i, G_i \rangle$ are *named graphs*, with u_i being the name of G_i. We assume that every dataset $\mathscr{D}$ is equipped with a function $d_{\mathscr{D}}$ such that $d_{\mathscr{D}}(u) = G$ if $\langle u, G \rangle \in \mathscr{D}$ and $d_{\mathscr{D}}(u) = \emptyset$ otherwise. Additionally, $\text{name}(\mathscr{D})$ stands for the set of IRIs that are names of graphs in $\mathscr{D}$, and $\text{term}(\mathscr{D})$ and $\text{blank}(\mathscr{D})$ stand for the set of terms and blank nodes appearing in the graphs of $\mathscr{D}$, respectively. For the sake of simplicity, and without loss of generality, we assume that the graphs in a dataset have disjoint sets of blank nodes, i.e. for $i \neq j$, $\text{blank}(G_i) \cap \text{blank}(G_j) = \emptyset$.

As we have seen in the previous section, the official syntax of SPARQL [15] considers operators `GRAPH`, `OPTIONAL`, `UNION`, `FILTER` and *conjunction* via a point symbol (`.`). The syntax also considers `{ }` to group patterns, and some implicit rules of precedence and association. For example, the point symbol (`.`) has precedence over `OPTIONAL`, and `OPTIONAL` is left associative. In order to avoid ambiguities in the parsing, in this section we present the syntax of SPARQL graph patterns in a more traditional algebraic formalism, using binary operators AND (`.`), UNION (`UNION`), OPT (`OPTIONAL`), FILTER (`FILTER`), and GRAPH (`GRAPH`). We fully parenthesize expressions making explicit the precedence and association of operators.

To define the algebraic syntax of SPARQL, we need to introduce the notions of *triple pattern* and *basic graph pattern*. A triple pattern is a tuple $t \in (I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$, and a basic graph pattern is a finite set of triple patterns. Notice that a triple pattern is essentially an RDF triple with some positions replaced by variables. Also notice that in our definitions of triple and basic graph pattern, we are not considering blank nodes. We make this simplification here to focus on the pattern matching part of the language. In Section 1.4.1, we discuss how these definitions should be extended to deal with blank nodes in basic graph patterns.

We use basic graph patterns as the base case for the syntax of SPARQL graph pattern expressions. A SPARQL graph pattern expression is defined recursively as follows:

1. A basic graph pattern is a graph pattern.
2. If P_1 and P_2 are graph patterns, then expressions $(P_1 \text{ AND } P_2)$, $(P_1 \text{ OPT } P_2)$, and $(P_1 \text{ UNION } P_2)$ are graph patterns (*conjunction graph pattern*, *optional graph pattern*, and *union graph pattern*, respectively).
3. If P is a graph pattern and $X \in I \cup V$, then $(X \text{ GRAPH } P)$ is a graph pattern.
4. If P is a graph pattern and R is a SPARQL *built-in* condition, then the expression $(P \text{ FILTER } R)$ is a graph pattern (a *filter graph pattern*).

A SPARQL *built-in* condition is constructed using elements of the set $I \cup L \cup V$ and constants, logical connectives ($\neg$, $\wedge$, $\vee$), ordering symbols ($<$, $\leq$, $\geq$, $>$), the equality symbol ($=$), unary predicates like bound, isBlank, and isIRI, plus other features (see [15] for a complete list). In this chapter, we restrict to the fragment of SPARQL where a built-in condition is a Boolean combination of terms constructed by using $=$ and bound, that is:

1. If $?X, ?Y \in V$ and $c \in I \cup L$, then $\text{bound}(?X)$, $?X = c$ and $?X =?Y$ are (atomic) built-in conditions.
2. If R_1 and R_2 are built-in conditions, then $(\neg R_1)$, $(R_1 \vee R_2)$ and $(R_1 \wedge R_2)$ are built-in conditions.

Let P be a SPARQL graph pattern. In the rest of this chapter, we use $\text{var}(P)$ to denote the set of variables occurring in P. In particular, if P is a basic graph pattern, then $\text{var}(P)$ denotes the set of variables occurring in the triple patterns that form P. Similarly, for a built-in condition R, we use $\text{var}(R)$ to denote the set of variables occurring in R.

We conclude the definition of the algebraic framework by describing the formal syntax of the SELECT query result form. A SELECT SPARQL query is simply a tuple (W, P), where P is a SPARQL graph pattern expressions and W is a set of variables such that $W \subseteq \text{var}(P)$.

1.3.1 Translating SPARQL *into the algebraic formalism*

In this section, we show that every SPARQL query can be translated into the algebraic terminology introduced above. But before providing the procedure that performs this translation, we show how the examples of Section 1.2 can be written in the algebraic formalism.

Example 1.5. First, consider the query "Give the name and the mailbox of each person who has a mailbox with domain .cl" from Example 1.1. The following algebraic expression represents this query (when evaluated over the RDF graph `myDataSource.rdf`):

```
({?name, ?mbox},
   ((?x, http://xmlns.com/foaf/0.1/name, ?name) AND
    (?x, http://xmlns.com/foaf/0.1/mbox, ?mbox) AND
    (?mbox, http://example.com/ns#domain, ".cl")))
```

Second, consider the query "Give the name and the mailbox, if it is provided, of each person in the FOAF file of Bob", which was considered in Example 1.2. The following algebraic expression represents this query (when evaluated over the graph `http://example.org/foaf/bobFoaf`):

```
({?name, ?mbox},
   ((?x, http://xmlns.com/foaf/0.1/name, ?name) OPT
    (?x, http://xmlns.com/foaf/0.1/mbox, ?mbox)))
```

Third, consider the query: "For every person known by Alice and Bob, give the nicknames by which are known by Alice and Bob" from Example 1.3. This query can be expressed as follows in the algebraic formalism:

```
({?nickA, ?nickB},
   ((http://example.org/foaf/bobFoaf GRAPH
       ((?x, http://xmlns.com/foaf/0.1/knows, ?comm) OPT
        (?comm, http://xmlns.com/foaf/0.1/nick, ?nickB))) AND
    (http://example.org/foaf/aliceFoaf GRAPH
       ((?y, http://xmlns.com/foaf/0.1/knows, ?comm) OPT
        (?comm, http://xmlns.com/foaf/0.1/nick, ?nickA)))))
```

Finally, consider the query: "Give the list of people for whom Alice knows at least two nicknames" from Example 1.4. The following expression represents this query (when evaluated over the graph `http://example.org/foaf/aliceFoaf`):

```
({?y},
   (((?x, http://xmlns.com/foaf/0.1/knows, ?y) AND
     (?y, http://xmlns.com/foaf/0.1/nick, ?nick1) AND
     (?y, http://xmlns.com/foaf/0.1/nick, ?nick2))
                              FILTER (?nick1 != ?nick2)))
```

□

In Algorithm 1, we show a transformation function $\mathscr{T}$ of patterns in the SPARQL syntax into the algebraic formalism presented in this section. For the sake of readability, we assume that the translation of Triple Blocks (`TB`) is given (this translation is straightforward, but tedious due to the multiple representations of triples allowed in the SPARQL syntax).

Algorithm 1 Transformation $\mathscr{T}$ of SPARQL pattern syntax into algebraic syntax

```
// Input: a SPARQL graph pattern GroupGP
// Output: an algebraic expression $E = \mathscr{T}(\texttt{GroupGP})$
$E \leftarrow \emptyset$; $FS \leftarrow \emptyset$
for each syntactic form $f$ in GroupGP do
    if $f$ is TB then $E \leftarrow (E$ AND $\mathscr{T}(\texttt{TB}))$
    if $f$ is OPTIONAL $\texttt{GroupGP}_1$ then $E \leftarrow (E$ OPT $\mathscr{T}(\texttt{GroupGP}_1))$
    if $f$ is $\texttt{GroupGP}_1$ UNION $\cdots$ UNION $\texttt{GroupGP}_n$ then
        if $n > 1$ then $E' \leftarrow (\mathscr{T}(\texttt{GroupGP}_1)$ UNION $\cdots$ UNION $\mathscr{T}(\texttt{GroupGP}_n))$
        else $E' \leftarrow \mathscr{T}(\texttt{GroupGP}_1)$
        $E \leftarrow (E$ AND $E')$
    if $f$ is GRAPH VarOrIRIref $\texttt{GroupGP}_1$ then
        $E \leftarrow (E$ AND (VarOrIRIref GRAPH $\mathscr{T}(\texttt{GroupGP}_1)))$
    if $f$ is FILTER constraint then $FS \leftarrow (FS \wedge \texttt{constraint})$
end for
if $FS \neq \emptyset$ then $E \leftarrow (E$ FILTER $FS)$
```

For example, consider the following pattern written according to the official SPARQL syntax:

```
{
    ?x :age ?y
    FILTER (?y > 30)
    ?x :knows ?z .
    ?z :home_country ?c
    FILTER (?c = "Chile")
    OPTIONAL  {  ?z :phone ?p  }
}
```

Following the grammar of SPARQL given in Fig. 1.2 the above pattern is parsed as a single `GroupGP` that contains the syntactic forms `TB`, `Filter`, `TB`, `Filter`, and `OptionalGP` in that order. This final `OptionalGP` syntactic form contains a `GroupGP` with a single `TB` syntactic form.

The translation function in Algorithm 1 starts with $E = \{\}$ and $FS = \{\}$. Then we consider all the syntactic forms in the pattern to obtain:

$$E = \Bigg(\Bigg(\big(\{\} \text{ AND } \mathscr{T}(\texttt{TB}_1) \big) \text{ AND } \mathscr{T}(\texttt{TB}_2) \Bigg) \text{ OPT } \mathscr{T}(\texttt{GroupGP}_1) \Bigg)$$

$$FS = \big((?Y > 30) \wedge ?C = \text{Chile} \big),$$

where $\texttt{TB}_1$ is `?x :age ?y`, $\texttt{TB}_2$ is `?x :knows ?z . ?z :home_country ?c`, and $\texttt{GroupGP}_1$ is `{ ?z :phone ?p }`. The translations $\mathscr{T}(\texttt{TB}_1)$ and $\mathscr{T}(\texttt{TB}_2)$ are simply $\{(?X, \text{:age}, ?Y)\}$ and $\{(?X, \text{:knows}, ?Z), (?Z, \text{:home\_country}, ?C)\}$, respectively.

To compute $\mathscr{T}(\texttt{GroupGP}_1)$ the algorithm proceeds recursively and gives as output the pattern:

$$E' = \big(\ \{\}\ \text{AND}\ \{(?Z, \text{:phone}, ?P)\}\ \big).$$

Finally the pattern in the algebraic syntax is:

$$\Bigg[\Bigg(\Bigg(\big(\ \{\}\ \text{AND}\ \{(?X, \text{:age}, ?Y)\}\ \big) \\ \text{AND}\ \{(?X, \text{:knows}, ?Z), (?Z, \text{:home\_country}, ?C)\}\Bigg) \\ \text{OPT}\ \big(\ \{\}\ \text{AND}\ \{(?Z, \text{:phone}, ?P)\}\ \big)\Bigg) \\ \text{FILTER}\ \big(\ (?Y > 30) \wedge ?C = \text{Chile}\ \big)\Bigg].$$

1.4 Semantics of SPARQL

To define the semantics of SPARQL graph pattern expressions, we use the algebraic representation of SPARQL introduced in the previous section.

We start by introducing some terminology. A *mapping* μ from V to T is a partial function $\mu : V \to T$. The domain of μ, denoted by $\text{dom}(\mu)$, is the subset of V where μ is defined. The empty mapping $\mu_\emptyset$ is a mapping such that $\text{dom}(\mu_\emptyset) = \emptyset$ (i.e. $\mu_\emptyset = \emptyset$). Given a triple pattern t and a mapping μ such that $\text{var}(t) \subseteq \text{dom}(\mu)$, $\mu(t)$ is the triple obtained by replacing the variables in t according to μ. Similarly, given a basic graph pattern P and a mapping μ such that $\text{var}(P) \subseteq \text{dom}(\mu)$, we have that $\mu(P) = \bigcup_{t \in P}\{\mu(t)\}$, i.e. $\mu(P)$ is the set of triples obtained by replacing the variables in the triples of P according to μ.

We can now define the semantics for basic graph patterns as a function $[\![\cdot]\!]_G$ that given a basic graph pattern P returns a set of mappings.

Definition 1.2. Let G be an RDF graph, and P a basic graph pattern. The *evaluation* of P over G, denoted by $[\![P]\!]_G$, is defined as the set of mappings

$$[\![P]\!]_G = \{\mu : V \to T \mid \text{dom}(\mu) = \text{var}(P) \text{ and } \mu(P) \subseteq G\}.$$

□

Notice that for every RDF graph G, it holds that $[\![\{\ \}]\!]_G = \{\mu_\emptyset\}$, i.e. the evaluation of an empty basic graph pattern against any graph results in the set containing only the empty mapping. For every basic graph pattern $P \neq \{\ \}$, we have that $[\![P]\!]_\emptyset = \emptyset$.

To define the semantics of more complex patterns, we need to introduce some more notions. Two mappings μ_1 and μ_2 are *compatible* when for all $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, it is the case that $\mu_1(x) = \mu_2(x)$, i.e. when $\mu_1 \cup \mu_2$ is also a mapping. Intuitively, μ_1 and μ_2 are compatible if μ_1 *can be extended* with μ_2 to obtain a new mapping, and vice versa. Note that two mappings with disjoint domains are always compatible and that the empty mapping $\mu_\emptyset$ is compatible with every other mapping.

Let Ω_1 and Ω_2 be sets of mappings. We define the join of, the union of and the difference between Ω_1 and Ω_2 as:

$$\begin{aligned}
\Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1, \mu_2 \text{ are compatible mappings}\},\\
\Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},\\
\Omega_1 \smallsetminus \Omega_2 &= \{\mu \in \Omega_1 \mid \text{ for all } \mu' \in \Omega_2,\ \mu \text{ and } \mu' \text{ are not compatible}\}.
\end{aligned}$$

Based on the previous operators, we define the left outer-join as:

$$\Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \smallsetminus \Omega_2).$$

Intuitively, $\Omega_1 \bowtie \Omega_2$ is the set of mappings that result from extending mappings in Ω_1 with their compatible mappings in Ω_2, and $\Omega_1 \smallsetminus \Omega_2$ is the set of mappings in Ω_1 that cannot be extended with any mapping in Ω_2. The operation $\Omega_1 \cup \Omega_2$ is the usual set theoretical union. A mapping μ is in $\Omega_1 ⟕ \Omega_2$ if it is the extension of a mapping of Ω_1 with a compatible mapping of Ω_2, or if it belongs to Ω_1 and cannot be extended with any mapping of Ω_2. These operations resemble the relational algebra operations but over sets of mappings (partial functions).

We are ready to define the semantics of SPARQL graph pattern expressions as a function $[\![\cdot]\!]_G^{\mathscr{D}}$ which given a dataset $\mathscr{D}$ and a (target) graph G in $\mathscr{D}$, takes a pattern expression and returns a set of mappings. For the sake of readability, the semantics of filter expressions is presented in a separate definition.

Definition 1.3. Let $\mathscr{D}$ be an RDF dataset and G an RDF graph in $\mathscr{D}$. The *evaluation* of a graph pattern P over G in the dataset $\mathscr{D}$, denoted by $[\![\cdot]\!]_G^{\mathscr{D}}$, is defined recursively as follows:

1. if P is a basic graph pattern, then $[\![P]\!]_G^{\mathscr{D}} = [\![P]\!]_G$.
2. if P is $(P_1 \text{ AND } P_2)$, then $[\![P]\!]_G^{\mathscr{D}} = [\![P_1]\!]_G^{\mathscr{D}} \bowtie [\![P_2]\!]_G^{\mathscr{D}}$.
3. if P is $(P_1 \text{ OPT } P_2)$, then $[\![P]\!]_G^{\mathscr{D}} = [\![P_1]\!]_G^{\mathscr{D}} ⟕ [\![P_2]\!]_G^{\mathscr{D}}$.
4. if P is $(P_1 \text{ UNION } P_2)$, then $[\![P]\!]_G^{\mathscr{D}} = [\![P_1]\!]_G^{\mathscr{D}} \cup [\![P_2]\!]_G^{\mathscr{D}}$.
5. if P is $(X \text{ GRAPH } P_1)$, then:
   - if $X \in I$, then $[\![P]\!]_G^{\mathscr{D}} = [\![P_1]\!]_{d_{\mathscr{D}}(X)}^{\mathscr{D}}$,
   - if $X \in V$, then
   $$[\![P]\!]_G^{\mathscr{D}} = \bigcup_{v \in \text{name}(\mathscr{D})} \left([\![P_1]\!]_{d_{\mathscr{D}}(v)}^{\mathscr{D}} \bowtie \{\mu_{X \to v}\} \right),$$
   where $\mu_{X \to v}$ is a mapping such that $\text{dom}(\mu) = \{X\}$ and $\mu(X) = v$.

Given a dataset $\mathscr{D}$ with default graph G_0, and a SPARQL pattern P, we say that the evaluation of P over dataset $\mathscr{D}$, denoted by $[\![P]\!]^{\mathscr{D}}$, is simply $[\![P]\!]_{G_0}^{\mathscr{D}}$. □

The idea behind the OPT operator is to allow for *optional matching* of patterns. Consider pattern expression $(P_1 \text{ OPT } P_2)$ and let μ_1 be a mapping in $[\![P_1]\!]_G^{\mathscr{D}}$. If there exists a mapping $\mu_2 \in [\![P_2]\!]^{\mathscr{D}}_G$ such that μ_1 and μ_2 are compatible, then $\mu_1 \cup \mu_2$ belongs to $[\![(P_1 \text{ OPT } P_2)]\!]^{\mathscr{D}}_G$. But if no such a mapping μ_2 exists, then μ_1 belongs to $[\![(P_1 \text{ OPT } P_2)]\!]^{\mathscr{D}}_G$. Thus, operator OPT allows information to be added to a mapping μ if the information is available, instead of just rejecting μ whenever some part of the pattern does not match. This feature of *optional matching* is crucial in Semantic Web applications, and more specifically in RDF data management, where it is assumed that every application has only partial knowledge about the resources being managed.

The operator GRAPH is used to change the *target* RDF graph over which a pattern is being evaluated. An expression of the form $(X \text{ GRAPH } P)$, with X an IRI and P a graph pattern, is used to change the target RDF graph to the one which name is X, and then to continue evaluating P over that RDF graph. The expression $(X \text{ GRAPH } P)$, with X a variable, is used to evaluate the pattern P over all the named RDF graphs in a dataset $\mathscr{D}$, and its result is the union of all these evaluations. Notice that before taking the union, for every $v \in \text{name}(\mathscr{D})$, the set of mappings obtained by evaluating pattern P over $d_{\mathscr{D}}(v)$ is joined with a mapping that assigns to variable X the value v. It should also be noticed that GRAPH is the only operator that can change the target RDF graph. Thus, if a pattern P does not contain any GRAPH expression, then the entire pattern is evaluated over a single RDF graph (the default graph of the RDF dataset). Therefore, if P is a SPARQL graph pattern expression that does not contain any GRAPH sub-expression, we simply write $[\![P]\!]_G$ to denote the set $[\![P]\!]^{\mathscr{D}}_G$. We use this notation in the following section when studying the complexity of evaluating graph pattern expressions.

The semantics of filter expressions goes as follows. Given a mapping μ and a built-in condition R, we define a notion of satisfaction of R by μ, denoted by $\mu \models R$, in a three valued logic (with values `true`, `false` and `error`}). For an atomic built-in condition of the form $?X = c$, if $?X \notin \text{dom}(\mu)$ the evaluation results in `error`; else, the evaluation results in `true` if $\mu(?X) = c$ and results in `false` otherwise. Similarly, for an atomic built-in condition of the form $?X = ?Y$, if $?X \notin \text{dom}(\mu)$ or $?Y \notin \text{dom}(\mu)$ the evaluation results in `error`; else, the evaluation results in `true` if $\mu(?X) = \mu(?Y)$ and results in `false` otherwise. For the case of bound$(?X)$, the evaluation results in `true` if $?X \in \text{dom}(\mu)$, and in `false` otherwise. For non-atomic constraints, the evaluation is defined as usual in a three valued logic:

R_1	R_2	$R_1 \wedge R_2$	$R_1 \vee R_2$
`true`	`true`	`true`	`true`
`true`	`error`	`error`	`true`
`true`	`false`	`false`	`true`
`error`	`true`	`error`	`true`
`error`	`error`	`error`	`error`
`error`	`false`	`false`	`error`
`false`	`true`	`false`	`true`
`false`	`error`	`false`	`error`
`false`	`false`	`false`	`false`

R_1	$\neg R_1$
`true`	`false`
`error`	`error`
`false`	`true`

.

Then $\mu \models R$ if and only if the evaluation of R against μ results in `true`.

Definition 1.4. Given an RDF dataset $\mathscr{D}$, an RDF graph G in $\mathscr{D}$, and a filter expression (P FILTER R), we have that $[\![(P \text{ FILTER } R)]\!]^{\mathscr{D}}_{G} = \{\mu \in [\![P]\!]^{\mathscr{D}}_{G} \mid \mu \models R\}$. □

Several algebraic properties of graph patterns are proved in [11]. A simple property is that AND and UNION are associative and commutative. This permits us to avoid parenthesis when writing sequences of AND operators or UNION operators. This is consistent with the definitions of Group Graph Pattern and Union Graph Pattern in [15]. To simplify the notation, when considering basic graph patterns composed by a single triple pattern $\{t\}$, we do not write the braces enclosing t. For example, for the pattern $((\{t_1\} \text{ UNION } \{t_2\}) \text{ OPT } \{t_3\})$, we simply write $((t_1 \text{ UNION } t_2) \text{ OPT } t_3)$. The following lemma shows that the base case for the syntax and semantics of SPARQL can also be defined in terms of triple patterns (instead of sets of triple patterns), as the semantics of basic graph patterns can be obtained by using the AND operator between triple patterns.

Lemma 1.1. *Let $\{t_1, t_2, \ldots, t_n\}$ be a basic graph pattern, where $n \geq 1$. Then for every dataset $\mathscr{D}$, it holds that:*

$$[\![\{t_1, t_2, \ldots, t_n\}]\!]^{\mathscr{D}} = [\![(t_1 \text{ AND } t_2 \text{ AND } \cdots \text{ AND } t_n)]\!]^{\mathscr{D}}.$$

To formally define the semantics of SELECT SPARQL queries, we need the following notion. Given a mapping $\mu : V \to T$ and a set of variables $W \subseteq V$, the *restriction* of μ to W, denoted by $\mu_{|_W}$, is a mapping such that $\text{dom}(\mu_{|_W}) = \text{dom}(\mu) \cap W$ and $\mu_{|_W}(?X) = \mu(?X)$ for every $?X \in \text{dom}(\mu) \cap W$.

Definition 1.5. Given a SELECT query (W, P), the evaluation of (W, P) in a dataset $\mathscr{D}$ is the set of mappings $[\![(W,P)]\!]^{\mathscr{D}} = \{\mu_{|_W} \mid \mu \in [\![P]\!]^{\mathscr{D}}\}$. □

In the rest of this chapter, we usually represent sets of mappings as tables where each row represents a mapping in the set. We label every row with the name of a mapping, and every column with the name of a variable. If a mapping is not defined for some variable, then we simply leave empty the corresponding position. For instance, the table:

	$?X$	$?Y$	$?Z$	$?V$	$?W$
μ_1 :	a	b			
μ_2 :		c			d
μ_3 :			e		

represents the set of mappings $\Omega = \{\mu_1, \mu_2, \mu_3\}$, where:

$\text{dom}(\mu_1) = \{?X, ?Y\}$, $\mu_1(?X) = a$, and $\mu_1(?Y) = b$,
$\text{dom}(\mu_2) = \{?Y, ?W\}$, $\mu_2(?Y) = c$, and $\mu_2(?W) = d$,
$\text{dom}(\mu_3) = \{?Z\}$, and $\mu_3(?Z) = e$.

We sometimes write $\{\{?X \to a, ?Y \to b\}, \{?Y \to c, ?W \to d\}, \{?Z \to e\}\}$ for a set of mappings as the one above.

Example 1.6. Consider an RDF graph G storing information about professors in a university:

$$
\begin{array}{lllllll}
G = \{ & (B_1, \text{name}, & \text{paul}), & (B_1, \text{phone}, & \text{777-3426}), & & \\
& (B_2, \text{name}, & \text{john}), & (B_2, \text{email}, & \text{john@acd.edu}), & & \\
& (B_3, \text{name}, & \text{george}), & (B_3, \text{webPage}, & \text{www.george.edu}), & & \\
& (B_4, \text{name}, & \text{ringo}), & (B_4, \text{email}, & \text{ringo@acd.edu}), & & \\
& (B_4, \text{webPage}, & \text{www.starr.edu}), & (B_4, \text{phone}, & \text{888-4537}) & \} &
\end{array}
$$

Let $\mathscr{D}$ be an RDF dataset with G as its default graph and with no named graphs. The following are graph pattern expressions and their evaluations over $\mathscr{D}$. Since the graph patterns do not use the GRAPH operator, we denote their evaluation by $[\![\cdot]\!]_G$.

- $P_1 = ((?A, \text{email}, ?E) \text{ OPT } (?A, \text{webPage}, ?W))$. Then

$[\![P_1]\!]_G =$

	$?A$	$?E$	$?W$
μ_1 :	B_2	john@acd.edu	
μ_2 :	B_4	ringo@acd.edu	www.starr.edu

- $P_2 = (((?A, \text{name}, ?N) \text{ OPT } (?A, \text{email}, ?E)) \text{ OPT } (?A, \text{webPage}, ?W))$. Then

$[\![P_2]\!]_G =$

	$?A$	$?N$	$?E$	$?W$
μ_1 :	B_1	paul		
μ_2 :	B_2	john	john@acd.edu	
μ_3 :	B_3	george		www.george.edu
μ_4 :	B_4	ringo	ringo@acd.edu	www.starr.edu

- $P_3 = ((?A, \text{name}, ?N) \text{ OPT } ((?A, \text{email}, ?E) \text{ OPT } (?A, \text{webPage}, ?W)))$. Then

$[\![P_3]\!]_G =$

	$?A$	$?N$	$?E$	$?W$
μ_1 :	B_1	paul		
μ_2 :	B_2	john	john@acd.edu	
μ_3 :	B_3	george		
μ_4 :	B_4	ringo	ringo@acd.edu	www.starr.edu

Notice the difference between $[\![P_2]\!]_G$ and $[\![P_3]\!]_G$. These two examples show that $[\![((A \text{ OPT } B) \text{ OPT } C)]\!]_G \neq [\![(A \text{ OPT } (B \text{ OPT } C))]\!]_G$ in general.

- $P_4 = ((?A, \text{name}, ?N) \text{ AND } ((?A, \text{email}, ?E) \text{ UNION } (?A, \text{webPage}, ?W)))$. Then

$[\![P_4]\!]_G =$

	$?A$	$?N$	$?E$	$?W$
μ_1 :	B_2	john	john@acd.edu	
μ_2 :	B_3	george		www.george.edu
μ_3 :	B_4	ringo	ringo@acd.edu	
μ_4 :	B_4	ringo		www.starr.edu

- $P_5 = (((?A, \text{name}, ?N) \text{ OPT } (?A, \text{phone}, ?P)) \text{ FILTER } (?N = \text{paul}))$. Then

$[\![P_5]\!]_G =$

	$?A$	$?N$	$?P$
μ_1 :	B_1	paul	777-3426

- $P_6 = (((?A, \text{name}, ?N) \text{ OPT } (?A, \text{phone}, ?P)) \text{ FILTER } (\neg\text{bound}(?P)))$. Then

$[\![P_6]\!]_G =$

	?A	?N	?P
μ_1 :	B_2	john	
μ_2 :	B_3	george	

□

The following example shows the evaluation of patterns that use operator GRAPH.

Example 1.7. Let G be the graph in Example 1.6 and consider the following RDF graph H:

$$\begin{aligned} H = \{ \; &(R_1, \text{name, mick}), \quad &&(R_1, \text{email, mj@acd.edu}), \\ &(R_2, \text{name, keith}), \quad &&(R_2, \text{email, keith@acd.edu}) \; \} \end{aligned}$$

Let $\mathscr{D} = \{\emptyset, \langle \text{tb}, G\rangle, \langle \text{trs}, H\rangle\}$ be an RDF dataset with empty default graph. The following are graph pattern expressions and their evaluations over $\mathscr{D}$.

- $P_7 = (\text{ trs GRAPH } (?A, \text{name}, ?N))$. Then

$[\![P_7]\!]^{\mathscr{D}} =$

	?A	?N
μ_1 :	R_1	mick
μ_2 :	R_2	keith

- $P_8 = (?G \text{ GRAPH } \{(?A, \text{name}, ?N), (?A, \text{email}, ?E)\})$. Then

$[\![P_8]\!]^{\mathscr{D}} =$

	?G	?A	?N	?E
μ_1 :	tb	B_2	john	john@acd.edu
μ_2 :	tb	B_4	ringo	ringo@acd.edu
μ_3 :	trs	R_1	mick	mj@acd.edu
μ_4 :	trs	R_2	keith	keith@acd.edu

□

Finally, the following example shows the evaluation of a SELECT pattern.

Example 1.8. Let $\mathscr{D}$ be the dataset in Example 1.7 and consider the pattern P_8 in that example. Then the evaluation of the SELECT query $(\{?G, ?N, ?E\}, P_8)$ over $\mathscr{D}$ is the following set of mappings:

$[\![(\{?G, ?N, ?E\}, P_8)]\!]^{\mathscr{D}} =$

	?G	?N	?E
μ_1 :	tb	john	john@acd.edu
μ_2 :	tb	ringo	ringo@acd.edu
μ_3 :	trs	mick	mj@acd.edu
μ_4 :	trs	keith	keith@acd.edu

□

1.4.1 Blank nodes in graph patterns

The official specification of SPARQL [15] allows basic graph patterns to have blank nodes in their triple patterns. Blank nodes in graph patterns are essentially defined as variables whose values cannot be retrieved by a query. In what follows, we extend the definitions of the previous sections to consider graph patterns with blank nodes.

We extend the definition of triple patterns to be tuples in the set $(T \cup V) \times (I \cup V) \times (T \cup V)$, that is, triple patterns are now allowed to have blank nodes as components. Similarly, we extend the definition of basic graph patterns. Also for a triple pattern t and a basic graph pattern P, we define $\text{blank}(t)$ and $\text{blank}(P)$ as the sets of blank nodes appearing in t and P, respectively.

Definition 1.6. Let G be an RDF graph and P a basic graph pattern with blank nodes. Then the evaluation of P over G, denoted by $[\![P]\!]_G$, is defined as the set of all mappings μ such that:

- $\text{dom}(\mu) = \text{var}(P)$,
- and there exists a substitution $\theta : \text{blank}(P) \to \text{term}(G)$ such that $\mu(\theta(P)) \subseteq G$,

where $\theta(P)$ is the basic graph pattern that results from replacing the blank nodes of P according to θ. □

This definition extends the definition of the semantics of a basic graph pattern P not mentioning blanks nodes, as by using the substitution $\theta : \emptyset \to \text{term}(G)$, we obtain the same set of mappings as in Definition 1.2 for pattern P (since $\theta(P) = P$).

Now, given a dataset $\mathscr{D}$ and a general graph pattern P constructed from basic graph patterns possibly with blank nodes, the evaluation of P over $\mathscr{D}$ is defined as in the previous section but with Definition 1.6 as the base case.

Example 1.9. Let G be the RDF graph in Example 1.6, and consider the basic graph pattern $P = \{(X, \text{name}, ?N), (X, \text{email}, ?E)\}$, where X is a blank node. Notice that, if we use a substitution $\theta : \text{blank}(P) \to \text{term}(G)$ such that $\theta(X) = B_2$, and a mapping $\mu = \{?N \to \text{john}, ?E \to \text{john@acd.edu}\}$, then we have that $\mu(\theta(P)) \subseteq G$. Thus, μ is in the evaluation of P over G. In fact, the evaluation of P over G is the set of mappings:

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

	$?N$	$?E$
μ_1 :	john	john@acd.edu
μ_2 :	ringo	ringo@acd.edu

□

1.4.2 Bag semantics of SPARQL

A *bag* Ω of mappings is a set of mappings in which every mapping is *annotated* with a positive integer that represents its *cardinality* in Ω. We denote the cardinality of the mapping μ in the bag Ω by $\mathrm{card}_{\Omega}(\mu)$ (or simply $\mathrm{card}(\mu)$ when Ω is understood from the context). If $\mu \notin \Omega$, then $\mathrm{card}_{\Omega}(\mu) = 0$.

In Section 1.4, we consider operations between sets of mappings. Those operations can be extended to bags by, roughly speaking, making the operations not to discard duplicates. Formally, if Ω_1, Ω_2 are bags of mappings, then:

$$\begin{aligned}
\text{for } \mu \in \Omega_1 \bowtie \Omega_2,\ \ \mathrm{card}_{\Omega_1 \bowtie \Omega_2}(\mu) &= \sum_{\mu = \mu_1 \cup \mu_2} \mathrm{card}_{\Omega_1}(\mu_1) \cdot \mathrm{card}_{\Omega_2}(\mu_2),\\
\text{for } \mu \in \Omega_1 \cup \Omega_2,\ \ \mathrm{card}_{\Omega_1 \cup \Omega_2}(\mu) &= \mathrm{card}_{\Omega_1}(\mu) + \mathrm{card}_{\Omega_2}(\mu),\\
\text{for } \mu \in \Omega_1 \smallsetminus \Omega_2,\ \ \mathrm{card}_{\Omega_1 \smallsetminus \Omega_2}(\mu) &= \mathrm{card}_{\Omega_1}(\mu).
\end{aligned}$$

The bag semantics of basic graph patterns that contain blank nodes is formalized in the following definition. This formalization is used as the base case for the bag semantics of SPARQL graph patterns.

Definition 1.7. Consider a basic graph pattern P (possibly with blank nodes) and an RDF graph G. The cardinality of the mapping $\mu \in [\![P]\!]_G$ is defined as the number of distinct substitutions $\theta : \mathrm{blank}(P) \to \mathrm{term}(G)$ such that $\mu(\theta(P)) \subseteq G$, i.e.

$$\mathrm{card}_{[\![P]\!]_G}(\mu) = |\{\theta : \mathrm{blank}(P) \to \mathrm{term}(G) \mid \mu(\theta(P)) \subseteq G\}|.$$

□

For a basic graph pattern P without blank nodes, every solution $\mu \in [\![P]\!]_G$ has cardinality 1, as in this case the only possible substitution is $\theta : \emptyset \to \mathrm{term}(G)$.

Given a dataset $\mathscr{D}$ and a general graph pattern P constructed from basic graph patterns possibly with blank nodes, we define the bag semantics of P over $\mathscr{D}$ simply as in Definition 1.3, but applying bag operators and considering the semantics of basic graph patterns as in Definition 1.7.

We define now the bag semantics of SPARQL SELECT queries. Informally, when considering bag semantics, to evaluate a SELECT query $q = (W, P)$, we simply take the projection of the evaluation of P over W but without discarding duplicates. Formally, given a SPARQL SELECT query (W, P) and a mapping μ in the evaluation of (W, P) over a dataset $\mathscr{D}$, we define the cardinality of μ in $[\![P]\!]^{\mathscr{D}}$ as:

$$\mathrm{card}_{[\![(W,P)]\!]^{\mathscr{D}}}(\mu) = \sum_{\nu \in [\![P]\!]^{\mathscr{D}} \,:\, \nu_{|W} = \mu} \mathrm{card}_{[\![P]\!]^{\mathscr{D}}}(\nu).$$

Example 1.10. Consider the RDF graph:

$$G = \{(\text{Alice, knows, Bob}), (\text{Alice, knows, Peter}), (\text{Bob, knows, Peter})\},$$

and the basic graph pattern $P = \{(?X,\ \text{knows},\ B)\}$ with B a blank node. Now consider the mapping $\mu_1 = \{?X \to \text{Alice}\}$, and the substitutions θ_1 and θ_2 from $\mathrm{blank}(P)$ to $\mathrm{term}(G)$ such that $\theta_1(B) = \text{Bob}$ and $\theta_2(B) = \text{Peter}$. Then it holds that $\mu_1(\theta_1(P)) \subseteq G$ and that $\mu_1(\theta_2(P)) \subseteq G$. Thus, we have that μ_1 is in $[\![P]\!]_G$ and that the cardinality of μ_1 is 2. If we consider the mapping $\mu_2 = \{?X \to \text{Bob}\}$, then we have that μ_2 is also in $[\![P]\!]_G$ and that the cardinality of μ_2 is 1. □

Example 1.11. As an example of the evaluation of a SELECT query under bag semantics, consider the dataset $\mathscr{D}$ and the pattern P_8 of Example 1.7, and the SELECT query $(\{?G\}, P_8)$. Then the evaluation of $(\{?G\}, P_8)$ over $\mathscr{D}$ is composed by the mappings $\mu_1 = \{?G \rightarrow \text{tb}\}$ and $\mu_2 = \{?G \rightarrow \text{trs}\}$, both with cardinality 2. □

1.5 On the Complexity of the Evaluation Problem

A fundamental issue in every query language is the complexity of query evaluation and, in particular, what is the influence of each component of the language in this complexity.

In this section, we study the complexity of the evaluation of SPARQL graph patterns, reviewing some of the results in the literature regarding this problem. The first study about the complexity of SPARQL was published in [11], and some refinements of the complexity results of [11] were presented in [13, 17]. We present here a study of the complexity that follows [11], considering fragments of SPARQL graph patterns built incrementally, and presenting complexity results for each such fragment.

In this section, we focus on the core fragment of SPARQL and, thus, we impose the following restrictions to graph patterns and to the evaluation process. First, we will be mainly focused on the evaluation of SPARQL patterns, that is, we do not consider SELECT queries, and we restrict to the evaluation over a single RDF graph, that is, we do not consider the GRAPH operator. Second, we assume that graph patterns do not contain blank nodes. And third, we focus on the set semantics of graph patterns, that is, we do not consider the cardinality of mappings when evaluating SPARQL patterns. It would be interesting to investigate whether the complexity results that we present in this section can be extended to the bag-semantics case. We left this study for future work.

As is customary when studying the complexity of the evaluation problem for a query language [18], we consider its associated decision problem. We denote this problem by EVALUATION and we define it as follows:

INPUT : An RDF graph G, a graph pattern P, and a mapping μ.
QUESTION : Is $\mu \in [\![P]\!]_G$?

It is important to recall that we are assuming that P in the above definition does not contain blank nodes, and that $[\![P]\!]_G$ is the set-based evaluation of P over the RDF graph G. Also notice that the evaluation problem that we study considers the mapping as part of the input. That is, we study the complexity by measuring how difficult it is to verify whether a given mapping is a solution for a pattern evaluated over an RDF graph. This is the standard *decision* problem considered when studying the complexity of a query language [18], as opposed to the *computation* problem of actually listing the set of solutions (finding all the mappings). To focus on the associated decision problem allows us to obtain a fine grained analysis of the complexity of the evaluation problem, classifying the complexity for different fragments of SPARQL in terms of standard complexity classes. Also notice that the pattern P and the graph G are both inputs in the definition of EVALUATION. Thus, we study the *combined complexity* of the query language [18].

We start this study by considering the fragment consisting of graph pattern expressions constructed by using only AND and FILTER operators. This simple fragment is interesting as it does not use the two most complicated operators in SPARQL, namely UNION and OPT. Given an RDF graph G, a graph pattern P in this fragment and a mapping μ, it is possible to efficiently check whether $\mu \in [\![P]\!]_G$ by using the following simple algorithm [11]. First, for each triple t in P, verify whether $\mu(t) \in G$. If this is not the case, then return *false*. Otherwise, by using a bottom-up approach, verify whether the expression generated by instantiating the variables in P according to μ satisfies the FILTER conditions in P. If this is the case, then return *true*, else return *false*.

Theorem 1.1 ([11]). EVALUATION *can be solved in time* $O(|P| \cdot |G|)$ *for graph pattern expressions constructed by using only* AND *and* FILTER *operators.*

We continue this study by adding the UNION operator to the AND-FILTER fragment. It is important to notice that the inclusion of UNION in SPARQL is one of the most controversial issues in the definition of this language. The following theorem proved in [11], shows that the inclusion of the UNION operator makes the evaluation problem for SPARQL considerably harder.

Theorem 1.2 ([11]). EVALUATION *is NP-complete for graph pattern expressions constructed by using only* AND, FILTER *and* UNION *operators.*

It is straightforward to prove that EVALUATION is in NP for the case of graph pattern expressions constructed by using only AND, UNION and FILTER operators. The NP-hardness proof presented in [11] relies on a reduction from the satisfiability problem for propositional formulas in CNF (SAT-CNF). An instance of SAT-CNF is a propositional formula φ of the form $C_1 \wedge \cdots \wedge C_n$, where each C_i ($i \in [1,n]$) is a clause, that is, a disjunction of propositional variables and negations of propositional variables. Then the problem is to verify whether there exists a truth assignment satisfying φ. It is well known that SAT-CNF is NP-complete [6]. In the encoding presented in [11], the authors use a fixed RDF graph D and a fixed mapping μ. Then they show how to encode a SAT-CNF formula by using SPARQL variables to encode literals (propositional variables and negations of propositional variables), AND and UNION to encode $\wedge$ and $\vee$, respectively, and FILTER restrictions to ensure that if a truth assignment assigns value true to a literal ℓ, then it must assign value false to the negation of ℓ (and vice versa).

We now consider the OPT operator, which is the most involved operator in graph pattern expressions and, definitively, the most difficult to define. The following theorem proved in [11] shows that when considering all the operators in SPARQL graph patterns, the evaluation problem becomes considerably harder.

Theorem 1.3 ([11]). EVALUATION *is PSPACE-complete.*

The membership in PSPACE is given by Algorithm 2. Given a mapping μ, a pattern P, and an RDF graph G, the algorithm verifies whether $\mu \in [\![P]\!]_G$. In the procedure, we use $\mathrm{pos}(P,G)$ to denote the set of mappings ν such that $\mathrm{dom}(\nu) \subseteq \mathrm{var}(P)$ and for every variable $?X \in \mathrm{dom}(\nu)$, it holds that $\nu(?X)$ is a value in $\mathrm{term}(G)$.

Algorithm 2 **Eval**(μ: mapping, P: graph pattern, G: RDF graph)

```
case:
P is a triple pattern t:
  if dom(μ) = var(t) and μ(t) ∈ G then return true
  return false
P is a pattern of the form (P_1 FILTER R):
  if Eval(μ, P_1, G) = true and μ ⊨ R then return true
  return false
P is a pattern of the form (P_1 UNION P_2):
  if Eval(μ, P_1, G) = true or Eval(μ, P_2, G) = true then return true
  return false
P is a pattern of the form (P_1 AND P_2):
  for each pair of mappings μ_1 ∈ pos(P_1, G) and μ_2 ∈ pos(P_2, G)
    if Eval(μ_1, P_1, G) = true and Eval(μ_2, P_2, G) = true and μ = μ_1 ∪ μ_2 then return true
  return false
P is a pattern of the form (P_1 OPT P_2):
  if Eval(μ, (P_1 AND P_2), G) = true then return true
  if Eval(μ, P_1, G) = true then
    for each mapping μ' ∈ pos(P_2, G)
      if Eval(μ', P_2, G) = true and μ is compatible with μ' then return false
    return true
  return false
```

It is easy to see that the procedure is correct (it is essentially applying the definition of the set-semantics of the SPARQL operators). Given that the size needed to store the name of a variable in $\mathrm{var}(P)$ is $O(\log|P|)$ and the size needed to store an element of G is $O(\log|G|)$, we obtain that the size of a mapping in $\mathrm{pos}(P,G)$ is $O(|P| \cdot (\log|P| + \log|G|))$. Thus, given that the depth of the tree of recursive calls to **Eval** is $O(|P|)$, we have that procedure **Eval** can be implemented by using a polynomial amount of space.

To prove the PSPACE-hardness of EVALUATION, the authors show in [11] how to reduce in polynomial time the quantified boolean formula problem (QBF) to EVALUATION. An instance of QBF is a quantified propositional formula φ of the form:

$$\forall x_1 \exists y_1 \forall x_2 \exists y_2 \cdots \forall x_m \exists y_m \, \psi,$$

where ψ is a quantifier-free formula of the form $C_1 \wedge \cdots \wedge C_n$, with each C_i ($i \in \{1, \ldots, n\}$) being a clause, that is, a disjunction of propositional variables and negations of propositional variables. Then the problem is to verify whether φ is valid. It is known that QBF is PSPACE-complete [6]. In the encoding presented in [11], the authors use a fixed RDF graph G and a fixed mapping μ. Then they encode formula

φ with a pattern P_φ that uses nested OPT operators to encode the *quantifier alternation* of φ, and a graph pattern not mentioning the optional operator to encode the satisfiability of formula ψ.

When verifying whether $\mu \in [\![P]\!]_G$, it is natural to assume that the size of P is considerably smaller than the size of G. This assumption is very common when studying the complexity of a query language. In fact, it is named *data complexity* in the database literature [18], and it is defined as the complexity of the evaluation problem for a fixed query. More precisely, for the case of SPARQL, given a graph pattern expression P, the evaluation problem for P, denoted by $\text{EVALUATION}(P)$, has as input an RDF graph G and a mapping μ, and the problem is to verify whether $\mu \in [\![P]\!]_G$. The following result shows that the data-complexity of the evaluation problem for SPARQL patterns is in LOGSPACE.

Theorem 1.4. $\text{EVALUATION}(P)$ *is in LOGSPACE for every graph pattern expression P.*

To see why the above theorem holds, consider Algorithm 2. The space needed to store a mapping in $\text{pos}(P,G)$ is $O(|P| \cdot (\log|P| + \log|G|))$, and this bound becomes $O(\log|G|)$ when P is considered to be fixed. Thus, given that the depth of the three of recursive calls to **Eval** is a fixed constant if P is considered to be fixed, we obtain that **Eval** can be implemented by using logarithmic space in this case.

1.6 Related Work

Most of the material presented in this chapter comes from [11]. At the time when [11] was published, there were two main proposals for the semantics of SPARQL graph pattern expressions. The first was an operational semantics, consisting essentially in the execution of a depth-first traversal of parse trees of graph pattern expressions, and the use of intermediate results to avoid some computations. At that time, this approach was followed by ARQ [2] (a language developed by HPLabs), and by the W3C when evaluating graph pattern expressions containing nested optionals [16]. For instance, the computation of the mappings satisfying $(A \text{ OPT } (B \text{ OPT } C))$ was done by first computing the mappings that match A, then checking which of these mappings match B, and for those that match B checking whether they also match C [16]. The second approach, compositional in spirit and the one advocated in [11], extended classical conjunctive query evaluation [7], and was based on a bottom up evaluation of parse trees of graph pattern expressions, borrowing notions of relational algebra evaluation [4, 8] plus some additional features. Currently, the official specification of SPARQL [15], endorsed by the W3C, formalizes a semantics based on [11], that we also follow in this chapter.

Since the beginning of the SPARQL standardization process by the W3C there have been efforts to formalize the semantics of the language. In [4], Cyganiak presents a relational model of SPARQL. The author uses modified versions of the standard relational algebra operators (join, left outer join, projection, selection, etc.)

to model SPARQL SELECT clauses. The central idea in [4] is to make a correspondence between SPARQL queries and relational algebra queries over a single relation *Triple*(*subject*, *predicate*, *object*), that stores RDF graphs in the form of triples. In [4], the author discusses some drawbacks of using classical relational algebra operators to define the semantics of SPARQL, and identifies cases in which his formalization does not match the SPARQL official specification. Additionally, a translation system between SPARQL and SQL is outlined in [4]. The system extensively uses COALESCE and IS NULL/IS NOT NULL operators to accurately resemble some SPARQL features. With different motivations, but similar philosophy, Harris et al. presents in [8] an implementation of a simple fragment of SPARQL in a relational database engine (they use relational algebra operators similar to the ones used in [4]).

As noted in [4], the treatment of null values is the major problem encountered when trying to specify the semantics of SPARQL by means of standard relational algebra. Since mappings must be modeled as relational tuples, null values need to be used to model unbounded variables. Zaniolo introduces in [19] an algebra to deal with null values in relational databases. The author interprets null values as standing for "no information", as opposed with the more complex "unknown" and "nonexistent" interpretations [9]. In [19], a *relation with null values* is defined as a set of tuples of not necessarily the same arity, which possibly contain null values in some of their components. The author then defines operators over those relations with nulls that generalize the standard relational algebra operators. The treatment of null values in [19] matches the treatment of unbounded variables in SPARQL. Thus, the operators over sets of mappings introduced in Section 1.3 can be easily modeled within the framework of [19]. Although the formalization in [19] can be used to define the semantics of SPARQL, we follow a simplified approach formalizing only what is strictly necessary in the SPARQL context, and thus simplifying the subsequent study of the language.

DeBruin et al. [3] study the semantics of the conjunctive fragment of SPARQL (graph patterns using only the AND operator, plus the SELECT clause) from a logical point of view. This semantics slightly differs from the definition in [15] on the issue of blank nodes. In their approach, blanks play the role of "non-distinguished" variables, that is, variables that are not presented in the answer.

In [14], Polleres studies the problem of translating SPARQL queries into Datalog queries. Based on [11], the author proposes three different semantics: (1) bravely-joining, (2) cautiously-joining, and (3) strictly-joining semantics. These semantics are obtained by strengthening the notion of compatible mappings, and thus, strengthening the conditions under which unbound variables are joined. Strictly-joining semantics essentially resembles the inner-join condition of SQL, allowing a simple translation into Datalog. Bravely-joining semantics coincides with the semantics presented in Section 1.4. To translate the bravely-joining semantics into Datalog, a special predicate $BOUND(\cdot)$ is needed to test whether a variable is bounded to a non-null value. As a result, the translation generates a program with disjunctions in the bodies of the rules that extensively uses $\neg BOUND(\cdot)$. The program is then transformed into Datalog by using standard techniques [14].

1.7 Conclusions

The query language SPARQL has been in the process of standardization since 2004. In this process, the semantics of the language has played a key role. A formalization of a semantics is beneficial on several grounds: help in identifying relationships among the constructors that stay hidden in the use cases, identify redundant and contradicting notions, study the expressiveness and complexity of the language, help in optimizing the evaluation of queries, etc. In this chapter, we have provided such a formal semantics for SPARQL, and we have reviewed some results concerning the complexity of evaluating SPARQL graph patterns.

References

1. R. Angles, C. Gutierrez. *The Expressive Power of SPARQL*. In *Proceedings of the Seventh International Semantic Web Conference*, pages 114–129, 2008.
2. ARQ. A SPARQL processor for jena, version 1.3 march 2006, hewlett-packard development company. `http://jena.sourceforge.net/ARQ`.
3. J de Bruijn, E. Franconi, S. Tessaris. *Logical reconstruction of normative rdf.* In *OWLED*.
4. R. Cyganiak. *A relational algebra for SPARQL.* Tech. Rep. HPL-2005-170, HP-Labs. `http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html`.
5. M. Durst and M. Suignard. *RFC 3987, Internationalized Resource Identifiers (IRIs).* `http://www.ietf.org/rfc/rfc3987.txt`.
6. M.R. Garey, D.S. Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1979. W. H. Freeman.
7. C. Gutierrez, C. Hurtado and A. Mendelzon. *Foundations of Semantic Web Databases*. In *Proceedings of the Twenty-third ACM Symposium on Principles of Database Systems (PODS)*, pages 95–106, 2004.
8. S. Harris, N. Shadbolt. *SPARQL query processing with conventional relational database systems.* In *WISE Workshops*. 235–244.
9. T. Imielinski, W. Lipski. *Incomplete information in relational databases. J. ACM 31,* 4, 761–791.
10. G. Klyne, J. J. Carroll and B. McBride. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation 10 February 2004. `http://www.w3.org/TR/rdf-concepts/`.
11. J. Pérez, M. Arenas and C. Gutierrez. *Semantics and Complexity of* SPARQL. In *Proceedings of the Fifth International Semantic Web Conference (ISWC)*, pages 30–43, 2006.
12. J. Pérez, M. Arenas and C. Gutierrez. *Semantics of SPARQL.* Technical Report, Universidad de Chile TR/DCC-2006-17, October 2006.
13. J. Pérez, M. Arenas and C. Gutierrez. *Semantics and Complexity of* SPARQL. To appear in *ACM Transaction on Database Systems*, 2009.
14. A. Polleres. *From SPARQL to rules (and back).* In *WWW*. 787–796.
15. E. Prud'hommeaux and A. Seaborne. *SPARQL Query Language for RDF*. W3C Recommendation 15 January 2008. `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`.
16. A. Seaborne. Personal communication, 2006.
17. M. Schmidt, M. Meier, G. Lausen. *Foundations of SPARQL Query Optimization*. arXiv.org paper arXiv:0812.3788v1, December 19, 2008.
18. M. Y. Vardi. *The Complexity of Relational Query Languages (Extended Abstract).* In *STOC 1982*, pages 137–146.
19. C. Zaniolo. *Database Relations with Null Values*. J. Comput. Syst. Sci. 28(1): 142–166 (1984).

Index