Published in final edited form as:

IEEE Trans Knowl Data Eng. 2009 March 1; 21(3): 401–414. doi:10.1109/TKDE.2008.91.

semQA: SPARQL with Idempotent Disjunction

E. Patrick Shironoshita¹, Yves R. Jean-Mary¹, Ray M. Bradley¹, and Mansur R. Kabuka²

¹ INFOTECH Soft, Inc., 9200 S. Dadeland Blvd., Suite 620, Miami, FL 33156. E-mail: patrick@infotechsoft.com, reggie@infotechsoft.com, rbradley@infotechsoft.com

² INFOTECH Soft, Inc., 9200 S. Dadeland Blvd., Suite 620, Miami, FL 33156 and the University of Miami, Coral Gables, FL 33124. E-mail: kabuka@infotechsoft.com

Abstract

The SPARQL `LeftJoin` abstract operator is not distributive over `Union`; this limits the algebraic manipulation of graph patterns, which in turn restricts the ability to create query plans for distributed processing or query optimization. In this paper, we present semQA, an algebraic extension for the SPARQL query language for RDF, which overcomes this issue by transforming graph patterns through the use of an idempotent disjunction operator `Or` as a substitute for `Union`. This permits the application of a set of equivalences that transform a query into distinct forms. We further present an algorithm to derive the solution set of the original query from the solution set of a query where `Union` has been substituted by `Or`. We also analyze the combined complexity of SPARQL, proving it to be NP-complete. It is also shown that the SPARQL query language is not, in the general case, fixed-parameter tractable. Experimental results are presented to validate the query evaluation methodology presented in this paper against the SPARQL standard to corroborate the complexity analysis and to illustrate the gains in processing cost reduction that can be obtained through the application of semQA.

Index Terms

Ontology languages; query languages; query processing

1 Introduction

The Resource Description Framework (RDF) is a World Wide Web Consortium (W3C) recommendation of a data format for representing information in the Web as directed labeled graphs [11]. Since its inception, several efforts have been made to define a language for querying RDF data sets. The Data Access Working Group within W3C has been engaged since 2004 in the definition of just such a query language, SPARQL, which has recently been approved as a recommendation [16].

Since the development of the relational model by Codd [2], query algebras have been defined to provide a formal basis for implementation, processing, and optimization of data sources and their querying mechanisms. Initial efforts at defining semantics and algebraic operations for RDF query languages in general include a navigational model defined on node sets [7], a model based on the application of relational algebra to the encoding of RDF as a single relation of three elements [3], and a definition stated from the perspective of mathematical logic and describing only basic graph patterns [1].

In a seminal and highly influential paper, Pérez et al. [13] presented a comprehensive and precise formalization of the semantics of the graph pattern matching fragment of SPARQL and an analysis of its complexity. The main premise of their approach rests on conceptualizing this graph pattern matching as the discovery of solution mappings between terms in an RDF data set and the variables in a graph pattern. The semantics of the language are then formalized as an algebra on operations between sets of these solution mappings. Pérez et al. also present an analysis of the complexity of SPARQL queries. Additionally, they have presented an extension to their semantics that formalize the concept of subgraph matching, using this definition to add support for blank nodes and multisets [14]. Some other extensions to the SPARQL semantics have been proposed in [15]. The SPARQL recommendation extends and formalizes the ideas proposed by Pérez et al., defining a set of abstract operators for the entire query language, establishing their semantics, and formalizing a query algebra [16].

In this paper, we present semQA, an extension to the SPARQL query algebra that uses an idempotent disjunction operator. We begin with a brief summary of the semantics of SPARQL based on the W3C recommendation [16]. Following this, we present a problem that hampers the algebraic manipulation of SPARQL queries, namely, the nondistributivity of the `LeftJoin` over the `Union` operator; this results in an inability to devise efficient query plans either for optimization of queries or for their distribution among multiple data sources. To overcome this, we introduce the idempotent disjunction `Or` operator, define the idempotent-disjunction (i-d) graph pattern by substituting `Union` with `Or`, and prove equivalences that permit the algebraic manipulation of an i-d graph pattern. Next, we present mechanisms for filter pushdown, that is, for the derivation of algebraically equivalent graph patterns that evaluate filters before other operations. We then present an algorithm to create the solution set for a graph pattern from the solution set of its corresponding i-d graph pattern. We also present an analysis of the complexity of SPARQL. First, we show that the data complexity of SPARQL queries is in P, while the combined complexity is NP-complete, and we show that neither the `Union` nor the `Or` operator add complexity to the query language. Next, we proceed to reanalyze the problem using parameterized complexity theory in terms of the size of the query, and we determine that there is strong evidence that SPARQL queries are not fixed-parameter tractable for the general case. Following this, we present experimental results that show the advantages gained through query manipulation and filter pushdown and that validate the complexity analysis presented. Finally, we present our conclusions and the direction of future work.

2 SPARQL Semantics and Algebra

In this section, we present a brief summary of the semantics and algebra of the SPARQL query language, as specified in the W3C recommendation [16], and we assert some algebraic equivalences that can be derived from these definitions.

2.1 Initial Definitions

An RDF graph G is a set of triples

$$t \in (I \cup B) \times I \times (I \cup B \cup L),$$

where I , B , and L are the sets of IRI references, blank nodes, and literals, respectively [11]. Table 1 shows an example RDF graph, where we have added identifiers for each triple for reference in the examples further below. Define the infinite set of RDF terms as $RDF-T = (I \cup B \cup L)$. We should indicate that any RDF graph G can be converted to a *canonical model* through the Skolemization of its blank nodes with some IRI not appearing elsewhere in G nor in the context in which it is used, i.e., in queries [1]. Every subsequent mention of RDF graphs

in this paper assumes a canonical model. An RDF data set D is a set $\{G_0, (u_1, G_1), \dots, (u_n, G_n)\}$, where each G_i is an RDF graph, and each u_i is an IRI; the graph G_0 is called the default graph, and each pair (u_i, G_i) is a named graph. One of the graphs in the data set, called the *active graph*, is used for basic graph pattern matching.

A *triple pattern* is a tuple

$$t_p \in (RDFT \cup V) \times (I \cup V) \times (RDFT \cup V),$$

where V is a set of variables disjoint from $RDF-T$. A *basic graph pattern* is a set of triple patterns. A *value constraint* is a Boolean-valued expression using elements from $(RDFT \cup V)$, logical functions, equality and in-equality symbols, and unary predicates. Let $P1$ and $P2$ be graph patterns and r be a value constraint; the SPARQL standard defines the following expressions: a *graph pattern with filter* $\{P1 \text{ FILTER } r\}$, a *group graph pattern* $\{P1\}\{P2\}$, an *alternative graph pattern* $\{P1\} \text{ UNION } \{P2\}$, and an *optional graph pattern* $\{P1\} \text{ OPTIONAL } \{P2\}$. Table 2 shows examples of these different patterns.

2.2 Solution Mappings and Solution Graphs

A *solution mapping* μ as a partial function $\mu: V \rightarrow RDF-T$. The domain of μ , that is, the set of variables upon which μ is defined, is denoted as $\text{dom}(\mu)$. A variable assigned a value through a solution mapping is said to be *bound* to the assigned value.

Blank nodes in a query pattern are treated as unnamed variables. To resolve blank nodes, a partial function called an *RDF instance mapping* $\sigma: B \rightarrow RDF-T$ is used. A *pattern instance mapping*¹ ρ is then defined as the combination of an RDF instance mapping and a solution mapping: $\rho: (V \cup B) \rightarrow RDF-T$, where $\rho(x) = \mu(x)$ if $x \in \text{dom}(\mu)$, and $\rho(x) = \sigma(x)$ if $x \in \text{dom}(\sigma)$. The domain of ρ , denoted as $\text{dom}(\rho)$, is given by the union of $\text{dom}(\mu)$ and $\text{dom}(\sigma)$.

Given a graph pattern P , then, $\rho(P)$ is the graph resulting from applying ρ to each variable and blank node in P ; for simplicity of notation, when there is no ambiguity we use ρ to denote both the pattern instance mapping and the resulting graph. Let $\text{var}(P)$ denote the set of variables and $\text{bn}(P)$ be the set of blank nodes in P ; then, ρ defines a unique solution mapping by restricting ρ to the variables of P , $\mu = \rho|_{\text{var}(P)}$, and an RDF instance mapping by similarly restricting ρ to the blank nodes of P , $\sigma = \rho|_{\text{bn}(P)}$. Also, define the domain of P , $\text{dom}(P) = \text{var}(P) \cup \text{bn}(P)$.

Two solution mappings μ_1 and μ_2 are said to be *compatible* if for every variable $v \in (\text{dom}(\mu_1) \cup \text{dom}(\mu_2))$, $\mu_1(v) = \mu_2(v)$. Similarly, two RDF instance mappings σ_1 and σ_2 are compatible if for every blank node $b \in (\text{dom}(\sigma_1) \cup \text{dom}(\sigma_2))$, $\sigma_1(b) = \sigma_2(b)$. Two pattern instance mappings are compatible if both the unique solution mappings and the unique RDF instance mappings they define are compatible.

SPARQL defines a solution mapping μ to be a *solution* for a basic graph pattern P from an RDF graph G if there exists a pattern instance mapping ρ such that $\rho(P)$ is a subgraph of G and $\mu = \rho|_{\text{var}(P)}$. We call the graph $\rho(P)$ a *solution graph* for P from G .

¹The SPARQL recommendation uses the symbol P for pattern instance matching; we use the symbol ρ to avoid confusion with graph patterns.

2.3 SPARQL Algebra Operators

A SPARQL abstract query is defined in the standard as a tuple $\{E, DS, RF\}$, where E is a SPARQL algebra expression, DS is an RDF data set, and RF is a query form. The algebra expression E in turn contains a graph pattern expression and solution modifiers.

The SPARQL standard defines graph pattern operators in terms of solution mappings. Here, we present these definitions in terms of solution graphs. Let Γ_1 and Γ_2 be multisets of solution graphs and let r be a value constraint. Denote $r(\rho)$ as the application of ρ to every variable in r . Then, we have the following:

1. $\text{Filter}(r, \Gamma_1) = \{\rho \mid \rho \in \Gamma_1, \text{ and } r(\rho) \text{ is true}\}.$
2. $\text{Join}(\Gamma_1, \Gamma_2) = \{\rho \mid \rho = \rho_1 \cup \rho_2, \rho_1 \in \Gamma_1, \rho_2 \in \Gamma_2, \text{ and } \rho_1 \text{ and } \rho_2 \text{ are compatible}\}.$
3. $\text{Diff}(\Gamma_1, \Gamma_2, r) = \{\rho \mid \rho \in \Gamma_1, \text{ and } \forall \rho_2 \in \Gamma_2, \text{ either } \rho \text{ and } \rho_2 \text{ are not compatible, or } r(\rho \cup \rho_2) \text{ is false}\}.$
4. $\text{LeftJoin}(\Gamma_1, \Gamma_2, r) = \text{Filter}(r, \text{Join}(\Gamma_1, \Gamma_2)) \cup \text{Diff}(\Gamma_1, \Gamma_2, r).$
5. $\text{Union}(\Gamma_1, \Gamma_2) = \{\rho \mid \rho \in \Gamma_1 \text{ or } \rho \in \Gamma_2\}.$

Proposition 1—The definitions given above in terms of solution graphs are equivalent to the definitions in terms of solution mappings in the SPARQL standard.

Proof—To prove this, we must show that 1) the evaluation of $r(\rho)$ is equivalent to the evaluation of $r(\mu = \rho|_{\text{var}(P)})$, and 2) that two pattern instance mappings ρ_1 and ρ_2 are compatible if and only if their corresponding solution mappings are compatible. Condition 1 follows immediately from the fact that no blank nodes are allowed in value constraints. Condition 2 is true because there are no blank nodes in common in the domains of ρ_1 and ρ_2 , as blank nodes are only scoped to the basic graph pattern.

2.4 Cardinality of Solutions

The SPARQL standard also defines the cardinality of solution mappings in each type of graph pattern. Then, in a similar manner as before, we present definitions in terms of solution graphs, derived from the original definitions by identifying $\mu = \rho|_{\text{var}(P)}$ for any graph pattern P . Let $\text{card}[\Gamma](\rho)$ denote the cardinality of a solution graph ρ in a solution multiset Γ , that is, the number of instances of ρ in Γ . Then, for Γ_1 and Γ_2 multisets of solution graphs and r value constraint, we have the following:

1. If Γ_1 is the solution multiset for a basic graph pattern P in an RDF graph G , $\text{card}[\Gamma_1](\rho)$ is 1.
2. $\text{card}[\text{Filter}(r, \Gamma_1)](\rho) = \text{card}[\Gamma_1](\rho).$
3. $\text{card}[\text{Join}(\Gamma_1, \Gamma_2)](\rho) = \text{for each } \rho_1 \in \Gamma_1, \rho_2 \in \Gamma_2 \text{ such that } \rho = \rho_1 \cup \rho_2, \text{ sum over } (\rho_1, \rho_2), \text{card}[\Gamma_1](\rho_1) * \text{card}[\Gamma_2](\rho_2).$
4. $\text{card}[\text{Diff}(\Gamma_1, \Gamma_2, r)](\rho) = \text{card}[\Gamma_1](\rho).$
5. $\text{card}[\text{LeftJoin}(\Gamma_1, \Gamma_2, r)](\rho) = \text{card}[\text{Filter}(r, \text{Join}(\Gamma_1, \Gamma_2))](\rho) + \text{card}[\text{Diff}(\Gamma_1, \Gamma_2, r)](\rho).$
6. $\text{card}[\text{Union}(\Gamma_1, \Gamma_2)](\rho) = \text{card}[\Gamma_1](\rho) + \text{card}[\Gamma_2](\rho).$

2.5 Evaluation Semantics

The expression $\text{eval}(G, P)$ denotes the evaluation of a graph pattern P over the active graph G of an RDF data set D . Let P, P_1 , and P_2 denote graph patterns and r be a value constraint; the following expressions define the evaluation of SPARQL graph patterns:

1. $\text{eval}(G, P)$, for P a basic graph pattern, results in the multiset of solution graphs for P .
2. $\text{eval}(G, \text{Filter}(r, P)) = \text{Filter}(r, \text{eval}(G, P))$.
3. $\text{eval}(G, \text{Join}(P_1, P_2)) = \text{Join}(\text{eval}(G, P_1), \text{eval}(G, P_2))$.
4. $\text{eval}(G, \text{LeftJoin}(P_1, P_2, r)) = \text{LeftJoin}(\text{eval}(G, P_1), \text{eval}(G, P_2), r)$.
5. $\text{eval}(G, \text{Union}(P_1, P_2)) = \text{Union}(\text{eval}(G, P_1), \text{eval}(G, P_2))$.

SPARQL also defines the evaluation semantics for *graph patterns on named graphs*, of the form $\text{Graph}(i, P)$ for $i \in I$ and $\text{Graph}(v, P)$ for $v \in V$, as well as the evaluation for expressions containing *solution modifiers*. These types of expressions do not lend themselves for the algebraic manipulation of graph patterns proposed in this paper and are thus not described further. It should be noted that the semantics of both the Graph operator and the solution modifiers, as described in the SPARQL standard, can be directly applied to the graph pattern expressions developed in this paper.

The eval operator notation is mainly necessary to specify the graph against which a graph pattern should be evaluated, which can vary only with the use of the Graph operator; therefore, where it is not ambiguous, we omit this notation and let P denote $\text{eval}(G, P)$ for any RDF graph G .

SPARQL queries written in the standard syntax are converted into the abstract queries through the application of rules specified in the recommendation. Table 3 shows the abstract query representations of the graph pattern portions of the queries in Table 2.

2.6 Algebraic Equivalences for Graph Patterns

Given two graph patterns P_1 and P_2 , we say that P_2 implies P_1 and denote it as $P_1 \leftarrow P_2$ if for every RDF graph G , every solution graph in $\text{eval}(G, P_2)$ is also in $\text{eval}(G, P_1)$. P_1 and P_2 are said to be equivalent, denoted as $P_1 \leftrightarrow P_2$, if they imply each other.

The SPARQL recommendation does not specify any implications or equivalences between graph pattern expressions. Nonetheless, some equivalences may be established from the operator definitions as follows:

Lemma 1—Let P_1 and P_2 be basic graph patterns; then, $P = (P_1 \cup P_2)$ is also a basic graph pattern.

Proposition 2—Given a basic graph pattern $P = (P_1 \cup P_2)$, $P \leftrightarrow \text{Join}(P_1, P_2)$.

Proof—Let ρ be a solution graph for P . Then, the restriction of ρ to $\text{dom}(P_1)$, $\rho|_{\text{dom}(P_1)}$, is a solution graph for P_1 and, similarly, for P_2 . Further, note that $\text{card}[P](\rho) = \text{card}[P_1](\rho|_{\text{dom}(P_1)}) = \text{card}[P_2](\rho|_{\text{dom}(P_2)}) = 1$. Note that this holds even if blank nodes are shared between P_1 and P_2 ; thus, the use of pattern instance mappings in the definitions allows us to define joins between intermediate solutions within basic graph patterns. Note also that the repeated application of this proposition yields the following:

Corollary 1—Given a basic graph pattern $P = \{t_1, t_2, \dots, t_n\}$, where each t_i is a triple pattern,

$$P \leftrightarrow \text{Join}(\dots \text{Join}(t1, t2), \dots), tn).$$

As a shorthand notation, we shall denote $\text{Join}(\dots \text{Join}(t1, t2), \dots), tn$ as $\text{Join}(t1, t2 \dots, tn)$.

Proposition 3—*Join is commutative and associative.*

Proof—The proof follows directly from the definition, considering that both set union and pattern instance mapping compatibility are commutative and associative.

3 semQA: SPARQL with Idempotent Disjunction

In this section, we introduce semQA as a modification on the standard SPARQL algebra. First, we show that LeftJoin is not distributive over Union . This results in an inability to perform meaningful algebraic manipulations over graph patterns. To overcome this, we introduce the idempotent disjunction Or operator, and define the i-d graph pattern by substituting Union with Or . We then state equivalences that permit the transformation of an i-d graph pattern; in particular, we define an equivalence to replace LeftJoin by a graph pattern combining Or and Join . We then present an algorithm to create the solution set for a graph pattern from the solution set of its corresponding i-d graph pattern.

3.1 Nondistributivity of LeftJoin over Union

Contrary to the assertion in [13, Proposition 1], LeftJoin is not distributive over Union .

Proposition 4—*Let $P1$, $P2$, and $P3$ be graph patterns and let r be a value constraint. $P = \text{LeftJoin}(P1, \text{Union}(P2, P3), r)$ is **not** equivalent to*

$$P' = \text{Union}(\text{LeftJoin}(P1, P2, r), \text{LeftJoin}(P1, P3, r)).$$

Proof—Suppose there exists a solution $\rho' \in \text{Filter}(r, \text{Join}(P1, P2))$ and suppose there does not exist another solution $\rho'' \in \text{Filter}(r, \text{Join}(P1, P3))$, where ρ' and ρ'' are compatible. Then, $\rho = \rho'|_{\text{dom}(P1)}$ is a solution for P' , since it is a solution for $\text{LeftJoin}(P1, P2, r)$. It is not a solution for P , however, since there exists a solution for $\text{Filter}(r, \text{Join}(P1, \text{Union}(P2, P3)))$, namely, ρ' . The proof is symmetric on $P2$ and $P3$.

This nonequivalence is illustrated in the example evaluation in Table 4, where P' results in one extra solution that matches $P1$ and where a compatible solution exists for $\text{Join}(P1, P2)$ but not for $\text{Join}(P1, P3)$.

Corollary 2—*It is not always possible to transform a graph pattern into an equivalent one of the form $\text{Union}(\text{Union}(P1, P2), \dots Pn)$, where each Pi is a graph pattern that does not contain Union operators.*

3.2 The Idempotent Disjunction Or

The inability to algebraically manipulate SPARQL expressions becomes a particular problem for distributed querying and query optimization: If queries cannot be transformed, it is not possible to devise query plans and select an efficient strategy either for querying of a single

RDF data source or for creating subqueries into multiple sources. One issue preventing the derivation of additional algebraic equivalences is the nonidempotence of the `Union` operator.

Proposition 5—*The `Union` operator is not idempotent, i.e., $\text{Union}(P1, P1)$ is not equivalent to $P1$.*

Proof—Let ρ be a solution for $P1$ with cardinality $\text{card}[P1](\rho)$. ρ is also a solution for $\text{Union}(P1, P1)$, with cardinality $2 * \text{card}[P1](\rho)$. Therefore, the solution multiset for $\text{Union}(P1, P1)$ contains exactly twice the number of solutions than the solution multiset for $P1$.

To surmount this obstacle, we introduce the `Or` operator.

Definition 1—*Let $P1$ and $P2$ be two graph patterns. We define the i-d graph pattern $\text{Or}(\Gamma_1, \Gamma_2)$ as follows:*

$$\text{Or}(\Gamma_1, \Gamma_2) = \{\rho \mid \rho \in \Gamma_1 \text{ or } \rho \in \Gamma_2, \text{ and there does not exist a } \rho' \in (\Gamma_1 \cup \Gamma_2) \text{ such that } \rho \subset \rho'\}.$$

The cardinality $\text{card}[\text{Or}(\Gamma_1, \Gamma_2)](\rho)$ is 1 for every ρ in $\text{Or}(\Gamma_1, \Gamma_2)$.

In this way, a graph ρ is a solution graph for $\text{Or}(P1, P2)$ over G if it is a solution graph for $\text{Union}(P1, P2)$ over G and if it is not a subgraph of any other solution graph for $\text{Or}(P1, P2)$. This definition makes `Or` an *idempotent disjunction*, since a solution graph ρ that is a solution for both $P1$ and $P2$ will only exist once in the set of solutions for $\text{Or}(P1, P2)$, and thus, $\text{Or}(P1, P1) \leftrightarrow P1$. Some examples illustrating the operation of `Or` are presented in Table 5. In the same way as with the `Join` operator, we denote $\text{Or}(\dots \text{Or}(P1, P2), \dots, Pn)$ with the shorthand $\text{Or}(P1, P2, \dots, Pn)$.

Proposition 6—*`Or` is commutative and associative.*

The proof follows directly from the definition.

The `LeftJoin` operator can now be expressed in terms of `Or` and `Join`.

Proposition 7—*For $P1$ and $P2$ graph patterns, $\text{LeftJoin}(P1, P2, r) \leftrightarrow \text{Or}(P1, \text{Filter}(r, \text{Join}(P1, P2)))$.*

Proof—Let

$$P = \text{LeftJoin}(P1, P2, r)$$

and

$$P' = \text{Or}(P1, \text{Filter}(r, \text{Join}(P1, P2))).$$

First, consider ρ to be a solution graph for $\text{Filter}(r, \text{Join}(P1, P2))$; by definition, this is a solution graph for P . It is also a solution for P' , since no ρ_x that is a solution of $P1$ could be a subgraph of ρ . Next, consider ρ' to be a solution for P so that it is a solution for $P1$ and there does not exist any ρ_y solution for $P2$ such that ρ' and ρ_y are compatible. Then, there does not

exist a solution for $\text{Join}(P1, P2)$ that is a superset of ρ' , and thus, ρ' is a solution for P' . Finally, consider ρ'' to be a solution for P so that it is a solution for $P1$, and for any solution ρ_z for $P2$, $r(\rho'' \cup \rho_z)$ is false. Then, $\text{Filter}(r, \text{Join}(P1, P2))$ does not have any solutions that are a superset of ρ'' , and thus, ρ'' is a solution for P' . Through similar reasoning, it can be shown that every solution for P' is also a solution for P , and thus, P and P' are equivalent patterns.

We propose to process SPARQL queries by replacing every instance of the `Union` operator with an `Or` operator; we call these *i-d graph patterns*. Using Proposition 7, any i-d graph pattern can then be transformed into one containing only `Join`, `Or`, and `Filter`. The solutions obtained through i-d graph patterns do not result in solutions conformant to the SPARQL standard; a simple correction algorithm is proposed further below to obtain conformant solutions. Before, some additional graph pattern implications and equivalences are presented.

Lemma 2—The multiset of solutions of any i-d graph pattern is actually a set.

Proof—`Or` does not allow a solution to exist more than once within its solution multiset. The solution multiset for $\text{Join}(P1, P2)$ is a set if the solution multisets of both $P1$ and $P2$ are sets. Similarly, the solution multiset for $\text{Filter}(r, P1)$ is a set if the solution multiset for $P1$ is a set.

Proposition 8—`Join` is idempotent for i-d graph patterns.

Proposition 9—Let $P1$, $P2$, and $P3$ be graph patterns and $R1$ and $R2$ be value constraints. Then,

$$\text{Join}(P1, \text{Or}(P2, P3)) \leftrightarrow \text{Or}(\text{Join}(P1, P2), \text{Join}(P1, P3)).$$

The proof for both these propositions follows directly from Lemma 2.

We can thus transform any i-d pattern P into an equivalent pattern $P' = \text{Or}(P_1, \dots, P_n)$, where each P_i does not contain any `Or` operators. Each P_i can in turn be transformed into an equivalent pattern $P'_i = \text{Join}(P_{i1}, \dots, P_{im})$, where each P_{jk} contains only `Join` operators. We call this the *disjunctive normal form* of an i-d graph pattern.

3.3 Filter Pushdown

One of the main strategies for optimization of relational queries consists of selection pushdown, that is, the ability to process conditions that constrain the size of result sets from queries at the earliest possible opportunity, avoiding costly processing of natural joins for tuples that will eventually be discarded. In an analogous fashion, in SPARQL, it is desirable to have the ability to push down filters. In this section, we propose a set of algebraic equivalences for filters within i-d graph patterns.

As in the SPARQL recommendation, we use the symbols `&&` and `||` to denote, respectively, *logical-and* and *logical-or* operations on value constraints. First, we present two equivalences that help in the resolution of these logical operators within value constraints.

Proposition 10—Let P denote a graph pattern and $r1$ and $r2$ denote value constraints. Then, we have the following:

$$1. \text{Filter}(r1 \ \&\& \ r2, P) \leftrightarrow \text{Filter}(r1, \text{Filter}(r2, P))$$

$$2. \text{Filter}(r_1 \ \&\& \ r_2, P) \leftrightarrow \text{Join}(\text{Filter}(r_1, P), \text{Filter}(r_2, P)).$$

Proof—Let ρ be a solution graph for P . If $r_2(\rho)$ is either false or an error, then ρ is not in the solution set of $\text{Filter}(r_1 \ \&\& \ r_2, P)$. It is also not in the solution set of $\text{Filter}(r_2, P)$ and thus not in the solution of the right-hand side of either expression. The proof is analogous for r_1 ; it is clear that both equivalences 1 and 2 are commutative on the logical and.

Proposition 11—Let P denote a graph pattern and r_1 and r_2 denote value constraints. Then,

$$\text{Filter}(r_1 \| r_2, P) \leftrightarrow \text{Or}(\text{Filter}(r_1, P), \text{Filter}(r_2, P)).$$

Proof—Let ρ be a solution graph for P . If at least one of $r_1(\rho)$ or $r_2(\rho)$ is true, then ρ is in the solution set of $\text{Filter}(r_1 \| r_2, P)$, and it is also in the solution of either $\text{Filter}(r_1, P)$ or $\text{Filter}(r_2, P)$.

One issue with the pushdown of filters through other operators is the handling of type errors when variables within a value constraint are not bound to a value. Consider, for example, the expression $\text{Filter}(r, \text{Join}(P_1, P_2))$; the objective is to evaluate the Filter before the Join , if possible. Now, suppose that r contains a variable in P_1 but not in P_2 ; then, it should only be evaluated against P_1 , since the evaluation of r against P_2 results in a type error.

We present here an algorithm to safely push down filters through Join and Or operators.

Definition 2—1) An atomic value constraint is a value constraint that does not include logical-and or logical-or operators. 2) A conjunctive value constraint is a value constraint composed of one or more atomic value constraints linked by logical-and operators. 3) A value constraint in disjunctive normal form consists of one or more conjunctive value constraints linked by logical-or operators.

Definition 3—Let $\text{var}(r)$ denote the set of variables in a value constraint r . Given a conjunctive value constraint r and a graph pattern P , a restriction of r on P , denoted $\mathbb{R}(r, P)$, is defined as follows:

1. If r is atomic, $\mathbb{R}(r, P) = r$ if $\text{var}(r) \subseteq \text{var}(P)$ and r is not of the form $!\text{bound}(v)$, true otherwise.
2. If $r = (r_1 \ \&\& \ r_2 \ \&\& \ \dots \ \&\& \ r_n)$, where every r_i is a conjunctive value constraint, we have the following:
 - a. $\mathbb{R}(r, P) = \text{false}$ if for all r_i , $\text{var}(r_i) \not\subseteq \text{var}(P)$ and r_i is not of the form $!\text{bound}(v)$.
 - b. $\mathbb{R}(r, P) = \mathbb{R}(r_1, P) \ \&\& \ \mathbb{R}(r_2, P) \ \&\& \ \dots \ \&\& \ \mathbb{R}(r_n, P)$.

Let $\mathbb{R}(r, P)(\rho)$ denote the evaluation of $\mathbb{R}(r, P)$ against a pattern instance mapping ρ .

Definition 4—Given a conjunctive value constraint r and graph patterns P_1 and P_2 , the overlap of r on P_1 and P_2 , denoted $\mathbb{L}(r, P_1, P_2)$, is defined as follows:

1. If r is atomic, $\mathbb{L}(r, P_1, P_2)$ is
 - a. r if it is of the form $!\text{bound}(v)$; or if $\text{var}(r) \subseteq (\text{var}(P_1) \cup \text{var}(P_2))$, and $\text{var}(r) \not\subseteq \text{var}(P_1)$ and $\text{var}(r) \not\subseteq \text{var}(P_2)$.

- b. false if $\text{var}(r) \not\subseteq (\text{var}(P1) \cup \text{var}(P2))$.
 - c. true if $\text{var}(r) \subseteq \text{var}(P1)$ or $\text{var}(r) \subseteq \text{var}(P2)$.
2. If $r = (r1 \ \&\& \ r2)$, for $r1$ and $r2$ conjunctive value constraints, $\mathbb{L}(r, P1, P2) = \mathbb{L}(r1, P1, P2) \ \&\& \ \mathbb{L}(r2, P1, P2)$.

Lemma 3—Let ρ_1 and ρ_2 be solution graphs for $P1$ and $P2$, respectively, let $\rho = \rho_1 \cup \rho_2$ be a solution graph for $P = \text{Join}(P1, P2)$, and let r be a conjunctive value constraint. If $r(\rho)$ produces an error and all variables in r are bound by ρ , then $\mathbb{L}(r, P1, P2)(\rho)$, $\mathbb{R}(r; P1)(\rho_1)$, or $\mathbb{R}(r; P2)(\rho_2)$ produce an error.

Proof—Suppose $r = r_1 \ \&\& \ \dots \ \&\& \ r_n$, and without loss of generality, assume that the error in r is produced by an error in a single r_k . If $\text{var}(r_k) \subseteq \text{var}(P1)$, then r_k is in $\mathbb{R}(r, P1)$, and thus, $\mathbb{R}(r, P1)(\rho_1)$ produces an error; the same goes for $P2$. Finally, if $\text{var}(r) \subseteq (\text{var}(P1) \cup \text{var}(P2))$, $\text{var}(r) \not\subseteq \text{var}(P1)$, and $\text{var}(r) \not\subseteq \text{var}(P2)$, $\mathbb{L}(r, P1, P2)(\rho)$ produces an error.

Lemma 4—Given a conjunctive value constraint r and i-d graph patterns $P1$ and $P2$,

$$\text{Filter}(r, \text{Join}(P1, P2)) = \text{Filter}(\mathbb{L}(r, P1, P2) \ \&\& \ \mathbb{R}(r, P1) \ \&\& \ \mathbb{R}(r, P2), \text{Join}(P1, P2)).$$

Proof—Let $P = \text{Filter}(r, \text{Join}(P1, P2))$,

$$P' = \text{Filter}(\mathbb{L}(r, P1, P2) \ \&\& \ \mathbb{R}(r, P1) \ \&\& \ \mathbb{R}(r, P2), \text{Join}(P1, P2))$$

and let $r = r_1 \ \&\& \ \dots \ \&\& \ r_n$, where every r_i is an atomic value constraint. First, suppose $\text{var}(r) \not\subseteq (\text{var}(\text{Join}(P1, P2)))$ and suppose that no r_i is of the form $! \text{bound}(v)$; then, $\text{var}(r) \not\subseteq (\text{var}(P1) \cup \text{var}(P2))$, so $\mathbb{L}(r, P1, P2)$ is false, and P' is empty. But, P is also empty since r will always produce an error. Also, note that by Lemma 3, any error on r not due to an unbound variable causes an error in $\mathbb{L}(r, P1, P2) \ \&\& \ \mathbb{R}(r, P1) \ \&\& \ \mathbb{R}(r, P2)$. Otherwise, every r_i in r is in $\mathbb{L}(r, P1, P2)$, $\mathbb{R}(r, P1)$, or $\mathbb{R}(r, P2)$, and thus, the filters are equivalent.

Proposition 12—Given i-d graph patterns $P1$ and $P2$ and a value constraint r , then

$$\begin{aligned} \text{Filter}(r, \text{Join}(P1, P2)) &\leftrightarrow \\ \text{Filter}(\mathbb{L}(r, P1, P2), & \\ \text{Join}(\text{Filter}(\mathbb{R}(r, P1), P1), \text{Filter}(\mathbb{R}(r, P2), P2))) & \end{aligned}$$

This is derived through the application of Lemma 4 and Proposition 10.

Proposition 13—Given i-d graph patterns $P1$ and $P2$ and a value constraint r , then

$$\begin{aligned} \text{Or}(\text{Filter}(r, P1), \text{Filter}(r, P2)) &\leftarrow \\ \text{Filter}(r, \text{Or}(P1, P2)) & \end{aligned}$$

Proof—Let ρ be a solution for $\text{Or}(P1, P2)$; then, it is either a solution for $P1$ or for $P2$. If $r(\rho)$ is true, then it is a solution for $\text{Filter}(r, \text{Or}(P1, P2))$ and also a solution for either $\text{Filter}(r, P1)$ or $\text{Filter}(r, P2)$. Note that the reverse implication is not necessarily true. Suppose some ρ' is a solution for $P1$, and ρ'' is a solution for $P2$ such that ρ' is a proper subgraph of ρ'' , and further suppose that $r = \text{!bound}(v)$ for some v bound for ρ'' but unbound for ρ' . Then, $r(\rho')$ is true, but $r(\rho'')$ is false. Thus, ρ' is a solution for $\text{Filter}(r, P1)$, ρ'' is not a solution for $\text{Filter}(r, P2)$, and therefore, ρ' is a solution for $\text{Or}(\text{Filter}(r, P1), \text{Filter}(r, P2))$. However, ρ' is not a solution for $\text{Or}(P1, P2)$, since ρ' is.

The identities shown here permit the definition of a method for filter pushdown within i-d graph patterns as follows:

1. First, convert all filter expressions to disjunctive normal form.
2. Next, apply Proposition 11 to resolve all *logical-or* expressions within filters. At this point, all filters contain only conjunctive value constraints.
3. Recursively apply Propositions 12 and 13 to push down filters through joins and ors until filters are acting directly on triple patterns.

In some cases, it may not be desirable or convenient to push down filters until they modify triples; strategies for query optimization must take into account metrics to let a system decide on an efficient query plan, in a manner analogous to the query optimization strategies used within relational databases. The ability to devise multiple query plans by algebraic manipulation and filter pushdown is necessary in order to design such query optimization strategies. The design of such strategies, including the use of cardinality estimation metrics such as those proposed in [17], is outside the scope of this paper and a matter for future work.

3.4 SPARQL Disjunction Correction Algorithm

The introduction of the Or operator proposed here enables the algebraic manipulation of SPARQL graph patterns. This provides a powerful tool for the optimization of queries, in combination with strategies to estimate the cardinality of partial results such as those proposed in [17]. The results obtained by this approach, however, are nonconformant to the current status of the SPARQL standard. We propose an algorithm to derive the results for a graph pattern P if the result set of its corresponding i-d graph pattern P' is known.

Consider an RDF graph G and a graph pattern P ; denote as $\Gamma(P, G)$ the set of solution graphs for P in G . Now, let P' be the i-d graph pattern derived from P . The objective is to build a SPARQL-conformant multiset $\Gamma(P, G)$ from the set of solutions $\Gamma'(P', G)$ of the i-d graph pattern. Note that by definition, any member of $\Gamma'(P', G)$ is a subgraph of G and, thus, an RDF graph in its own right. To build $\Gamma(P, G)$, we construct a disjunction correction parse tree representing P , where the leaf nodes are triple patterns t_{P_x} , and all other nodes are binary operators and unary value constraints. We then process each $\rho'_i \in \Gamma'(P', G)$ bottom-up through this parse tree, generating outputs from each node in the following manner:

- A leaf node with triple pattern t_{P_x} outputs solution set $\Gamma(t_{P_x}, \rho_i)$, which consists of either a single triple or the empty set $\emptyset$.
- A node containing a value constraint r outputs the subset of its input for which $r(\rho_i)$ is true.
- A Join node with inputs Γ_1 and Γ_2 outputs the cross product $\Gamma_1 \times \Gamma_2$. Observe that every element of such cross product is a subgraph of ρ'_i .
- A LeftJoin node with inputs Γ_1 and Γ_2 outputs Γ_1 if Γ_2 is $\emptyset$ and the cross product $\Gamma_1 \times \Gamma_2$ otherwise.

- A Union node with inputs Γ_1 and Γ_2 outputs $\Gamma_1 \cup \Gamma_2$.

All cross products and set unions are performed preserving duplicates. The root node of the parse tree will hold a partial solution $\Gamma_i(P, G)$. This process is repeated for every $\rho'_i \in \Gamma'(P', G)$. Let the multiset $\Gamma_u(P, G)$ be the union of all $\Gamma_i(P, G)$; we now show that all $\rho \in \Gamma(P, G)$ must also be in $\Gamma_u(P, G)$ as follows:

Lemma 5—Consider a set $H \subseteq \Gamma(P, G)$ such that all $h_j \in H$ are equal to some graph ρ ; the size of H is called the cardinality of ρ in $\Gamma(P, G)$. All $h_j \in H$ are subsets of only one $\rho' \in \Gamma'(P', G)$.

Proof—This follows from the fact that all h_j are generated by the application of Union operators within P . When the i-d graph pattern P' is derived from P , all Union operators are substituted by Or operators, and all multiple equivalent solutions at each subpattern result in a single solution by the definition of Or. Therefore, all h_j result in a single $\rho' \in \Gamma'(P', G)$.

Proposition 14—All ρ in $\Gamma(P, G)$ are also in $\Gamma_u(P, G)$.

Proof—From Lemma 5, every member of $\Gamma(P, G)$ is a subset of one and only one member of the solution set $\Gamma'(P', G)$, and thus, all $\rho \in \Gamma(P, G)$ will be in $\Gamma_u(P, G)$ and will exist in the same cardinality.

On the other hand, not all $\rho' \in \Gamma_u(P, G)$ are necessarily in $\Gamma(P, G)$. It is clear that every ρ' must be at least a subgraph of some $\rho \in \Gamma(P, G)$, since otherwise, ρ' would not be a result in the disjunction correction parse tree. The situation whereby ρ' is not in $\Gamma(P, G)$ arises from the elimination of subgraphs through the LeftJoin operator. The solution multiset $\Gamma_s(P, G)$ is then obtained as the multiset of graphs $\rho' \in \Gamma_u(P, G)$ such that either there does not exist a solution graph ρ'' for which $\rho' \subset \rho''$ or if it exists, then the graph difference between the two, $\rho'' \setminus \rho'$, is not a solution for the right-hand side of a LeftJoin operator. With this final correction and the result of Proposition 14, $\Gamma_s(P, G)$ only contains solutions for P , it contains every solution for P , and it contains each solution in the exact number of occurrences prescribed by the SPARQL standard. Therefore, we have the following:

Proposition 15—The SPARQL disjunction correction algorithm produces a SPARQL-conformant solution multiset.

An example of a parse tree resolution is graphically illustrated in Fig. 1. The SPARQL disjunction correction algorithm is depicted in pseudocode form in Table 6.

4 The Complexity of SPARQL Queries

In classical database theory, the query evaluation problem is analyzed from different points of view, depending on the parameters with respect to which the complexity is measured [18]. There are two fundamental perspectives: *data complexity*, which considers the query fixed and evaluates complexity with respect to the size of the database, and *combined complexity*, which evaluates complexity against the combined size of both the database and the query.

The SPARQL graph mapping query evaluation problem, which we will call SPARQL-EVALUATION for short, is the extraction of a set of subgraphs from an RDF graph G through the use of a graph pattern P . An analysis of the combined complexity of SPARQL is offered in [13], separating the discussion into three increasingly inclusive subsets: 1) conjunctive queries, which contain only Join and Filter operations, 2) positive queries, which add Union, and 3) all queries, which encompass the full SPARQL language including the

LeftJoin operator. The analysis in [13] states that these three subsets have an increasing complexity. However, a close verification of the proofs offered by the authors in the long version of their paper led us to reexamine these conclusions; in particular, we show here that the combined complexity of conjunctive queries is NP-complete and, thus, intractable.²

As a preliminary point, the evaluation of a value constraint r in an expression $\text{Filter}(r, P)$ can be performed in constant time once a solution for P is found, as long as the expressions in the value constraints are limited to the built-in functions defined in [16]; thus, the existence of Filter expressions restricted in this way does not affect their complexity and will be ignored in this analysis. SPARQL does allow user-defined functions, which could significantly affect the complexity of the overall query. We should also highlight that some suggestions, notably the ability to use ASK queries as value constraints in Filter expressions as suggested in [15], could also cause a significant increase in complexity and should be closely studied.

4.1 Data Complexity

In order to analyze the complexity of a problem, it is typical to reduce a function problem such as SPARQL-EVALUATION to a decision problem. To analyze the data complexity of query evaluation in relational databases, for example, the associated decision problem is to determine if a given tuple t belongs in the set of answers from a fixed query Q . Using this formulation, it is well known that the complexity of the relational algebra is between AC_0 and P [18].

In a similar fashion, given an RDF graph G , a graph pattern P , and a graph $\rho \subseteq G$, the data complexity for SPARQL queries over RDF can be determined by asking if $\rho \in \Gamma(P, G)$; we shall call this the SPARQL-D problem. Note that the possible number of ρ that could satisfy this problem is in $O(N^q)$, where N is size of the RDF graph measured by the number of triples, and q is the size of the pattern measured by the number of distinct triple patterns it contains; this is because every triple pattern in the query has at most N possible matches. This in turn means that the SPARQL-D problem can be reduced to the SPARQL-EVALUATION problem in polynomial time with respect to the size of the RDF graph. It follows that the SPARQL-D problem when considering a fixed query size is in AC_0 for conjunctive queries. In addition, by analogy to relational algebra, it can be seen that SPARQL-D is at most in P for all queries.

Much as in relational database queries, these results suggest that SPARQL queries are computationally quite simple to solve. However, note that in practice, while queries are generally orders of magnitude smaller than data sets, they are by no means fixed. The question then is: how does the query size affect the complexity of a SPARQL query? More importantly, for practical purposes, what limitations do we have on queries for them to be efficiently solvable? We attempt to answer these questions through the analysis of the combined complexity of these queries.

4.2 Combined Complexity

To analyze the combined complexity of SPARQL graph mapping, we must similarly determine an appropriate decision problem for SPARQL-EVALUATION but in terms of combined complexity. Pérez et al. [13] propose the decision problem as follows: Given a solution mapping μ , determine if this mapping belongs to the set of solutions for a graph pattern P in an RDF graph G . Note that this formulation is analogous to the SPARQL-D decision problem stated above but using solution mappings instead of pattern instance mappings. Then, if G contains N triples and P contains k variables, the verification of whether μ is a solution for P can be done in $O(Nk)$, that is, in linear time.

²As is usual in complexity theory, all assertions of intractability in this paper are made assuming that $\text{P} \neq \text{NP}$.

However, consider that the upper bound for the total number of possible mappings μ is N^k , since every variable in P could be assigned up to N values. Therefore, to find all possible solution mappings μ for P using this algorithm, each of the N^k possible solution mappings must be checked to see if it belongs to the set of solutions for P , and thus, the total resolution of the SPARQL-EVALUATION problem in this manner can only be done in $O(NkN^k)$, and the combined complexity of the conjunctive query solution algorithm in [13] is EXPTIME.

As this analysis shows, the SPARQL-D decision problem cannot be easily reduced to the evaluation problem when considering the size of the query as variable. An appropriate decision problem, which we shall call SPARQL-C, is to determine if a solution ρ exists for a graph pattern P over RDF graph G ; then, it can be seen that a solution to the graph mapping evaluation problem can be found in polynomial time by choosing at random a variable in P , assigning it to some term in G and asking if P with the variable replaced with the chosen term still has a solution. Following this, we classify the complexity of the different subsets of SPARQL, including the subset of idempotent-positive queries that include only `Join` and `Or` operators.

Theorem 1—*The SPARQL-C-CONJUNCTIVE problem is NP-complete.*

Proof—It is easy to see that the problem is in NP. Further, it can be readily observed that the problem is almost the SUBGRAPH-ISOMORPHISM problem for directed labeled graphs, which is known to be NP-complete. The “almost” is due to the fact that the pattern instance mapping ρ is not bijective, since two variables $v_1, v_2 \in \rho$ could be mapped to the same value. Thus, SUBGRAPH-ISOMORPHISM can be trivially reduced to SPARQL-C-CONJUNCTIVE by requiring that ρ be bijective.

Theorem 2—*The SPARQL-C-IDEMPOTENT-POSITIVE problem is NP-complete.*

Proof—To prove this assertion, first note that the problem is clearly in NP: A nondeterministic machine can guess a solution and verify it in polynomial time, as shown by the disjunction correction algorithm presented in the previous section. We then reduce from the Boolean circuit satisfiability problem, which is known to be NP-complete. Consider a circuit representing a Boolean expression $\phi(y_1 \dots y_m)$ and assume an RDF graph G , where each triple $t \in G$ contains either one of two values, a_1 or a_2 , at the property position and any value from the universe of RDF terms at the subject and object positions. Let $t_k = \{?v_{k1}a_1?v_{k2}\}$ and $t'_k = \{?v_{k1}a_2?v_{k2}\}$, where no two $?v_{ij}$ variables are the same; clearly, one and only one of t_k or t'_k has a solution on G . Then, build a graph pattern by replacing y_k with t_k and $\neg y_k$ with t'_k and by changing every Boolean conjunction for a `Join` operator and every Boolean disjunction for an `Or` operator. It follows that P will have a solution if and only if $\phi(y_1 \dots y_m)$ is satisfiable.

Corollary 3—*The SPARQL-C-POSITIVE problem is also NP-complete.*

Proof—This follows from Theorem 2 plus the fact that given a graph pattern P and its corresponding i-d graph pattern P' , if there exists a solution for P' , then there exists a solution for P . From this, the following also follows:

Corollary 4—*The SPARQL-C-ALL problem is also NP-complete.*

In [13], it is stated that SPARQL-C-ALL is PSPACE-complete. However, a close analysis of the proof offered by the authors in the extended version of their paper shows that a fixed database was used to prove completeness, thus deriving complexity only against the size of the query, that is, *expression complexity*, and not the combined complexity.

These results lead to the following important conclusion:

Corollary 5—*Neither the Union, the Or, nor the LeftJoin operator add any computational complexity to the problem of evaluating SPARQL queries that already contain conjunctions.*

From these results, we can conclude that SPARQL queries, in the same way as relational queries, quickly become complex as the size of the query grows. The SPARQL query evaluation problem is therefore intractable except for small sizes of databases and queries; this is in direct contradiction to the results of looking only at the data complexity.

However, neither of these two notions of complexity is completely satisfactory [12]. On the one hand, polynomial time in data complexity for queries means time in $O(N^q)$, where N is the size of the data set and q is the size of the query; this hardly qualifies as tractable, since in practice, the query size is not fixed. On the other hand, combined complexity treats both N and q in the same way, even though $q \ll N$; this assigns too much importance to q and thus classifies the evaluation problem as harder than it is. As stated in [12], parametric complexity is a productive framework for studying these problems, since there is a large main measure of the input (the size N of the database) and a small but significant parameter, the size of the query. In fact, one of the principal problem areas in which parameterized complexity has been applied is in the evaluation of relational database queries [6]. In the next section, we present an analysis to the parameterized complexity for SPARQL conjunctive and positive queries.

4.3 Parameterized Complexity

Parameterized complexity has been developed as a means to study computational problems that have inputs that consist of several elements of information, where some of these elements are considered parameters in order to evaluate their relative contribution to the overall complexity of the problem [5]. In particular, parametric problems with input N and parameter k are said to be *fixed-parameter tractable* if they can be solved in $O(f(k) * N)$, where $f(k)$ is an (possibly) exponential function; intractable problems are only solvable in $O(N^{g(k)})$. A sequence of complexity classes of parametric problems, called the W-hierarchy, has been defined to classify them; problems at the higher levels of this hierarchy are less likely to be fixed-parameter tractable [4], [6].

Query evaluation is a natural problem to be analyzed using parameterized complexity theory, where the natural parameter is the size of the query. Substantial work has been done in such evaluation for relational databases [5], [8], [12]. In this section, we present a parameterized analysis of SPARQL query evaluation and use these results to perform an initial examination of algorithms that can be used to solve queries efficiently.

In order to analyze the parameterized complexity of the SPARQL query evaluation problem, we reformulate it as follows: Given a graph pattern P of size q and an RDF graph G , determine if a solution ρ exists for P over G ; we now call this the p-SPARQL problem. Up to this point, we have assumed the size of the graph pattern P to be the number of elements (i.e., triple patterns and value constraints) in the pattern, but this size could also be measured by the number of its variables. This distinction does not alter the results for combined complexity, as can be seen from the proofs in the previous section. The parameterized complexity analysis for relational queries made in [12] does show some difference between the two measures of query size; in our case, the use of either the number of terms or the number of variables as a measure for query size produces the same complexity results.

Theorem 3—*The p-SPARQL-CONJUNCTIVE problem is W[1]-hard parameterized either by the number of variables or by the number of terms.*

Proof—It follows immediately from the fact that the parameterized subgraph isomorphism problem $p\text{-SUBGRAPH-ISOMORPHISM}$ is $W[1]$ -hard [9] and that this problem can be reduced to a special case of the $p\text{-SPARQL-CONJUNCTIVE}$ problem. Note also that reducing from $p\text{-SUBGRAPH-ISOMORPHISM}$, it can be shown that since RDF terms in the query graph pattern are mapped with themselves, the problem reduces to $p\text{-SUBGRAPH-ISOMORPHISM}$ on a graph pattern whose nodes are only variables. We note also that the $p\text{-SPARQL-ALL}$ problem is clearly at least $W[1]$ -hard, since it is a generalization of $p\text{-SPARQL-CONJUNCTIVE}$.

Corollary 6—SPARQL queries are not fixed-parameter tractable when parameterized either by the number of variables or by the number of terms in a query.

Note that this is analogous to the situation for relational queries, which are also fixed-parameter intractable in the general case but become fixed-parameter tractable for a particular class of queries, i.e., acyclic conjunctive queries [12]. We suggest that in SPARQL, it should also be possible to similarly find types of queries that become fixed-parameter tractable; in particular, we are exploring the behavior of *tree queries*, where graph patterns are trees. In addition, we believe that algorithms that evaluate queries more efficiently may be developed through the conceptualization of conjunctive queries as subgraph isomorphisms. We are currently exploring approaches to the SPARQL query evaluation problem based on the substantive research existing in graph theory.

5 Experimental Results

In order to validate the semQA algebra extension and the complexity analysis presented here, we developed a prototype implementation based on extensions to the source code of version 2.1 of the SPARQL query engine ARQ (<http://jena.sourceforge.net/ARQ/>). Our prototype parses and converts SPARQL queries into i-d graph patterns, uses ARQ methods to obtain the result set of triple pattern matching, resolves all `Join`, `Or`, and `Filter` operations with its own methods, and implements the disjunction correction algorithm. At its current implementation, it only works with `SELECT` queries, does not process solution modifiers, and does not work with `Graph` graph patterns.

With this prototype, we ran three sets of experiments: the suite of SPARQL test cases, to validate that the algorithm achieves correct results according to the standard; graph pattern evaluation against RDF graphs of different sizes, to measure the effect of algebraic manipulation and filter pushdown on the efficiency of query processing; and a set of graph pattern evaluations with varying numbers of triple patterns, to verify the results from the combined complexity analysis presented in this paper.

5.1 SPARQL Test Cases

The Data Access Working Group within W3C maintains a collection of test cases in <http://www.w3.org/2001/sw/DataAccess/tests/r2.html>. Of these tests, we ran those listed in Table 7; all other tests included features that are not yet implemented in our prototype. Nevertheless, the test selection includes a diverse set of queries using group, optional, and alternative graph patterns and filter expressions.

Our prototype passed all tests except for open-eq-1. ARQ also fails this test, which requires that `"01"^^xsd:integer` and `"001"^^xsd:integer` not to be considered equal due to the definition of entailment in SPARQL. For test expr-5, which has two possible sets of results, our algorithm produces the version with the simplification of curly braces applied during transformation.

These results show that to the extent that is implemented, the algorithm proposed here obtains results consistent with the SPARQL standard.

5.2 Experiments with Algebraic Manipulation and Filter Pushdown

In order to experimentally illustrate the influence of algebraic manipulation and filter pushdown on query processing cost, we ran tests of two graph pattern evaluations: one to evaluate algebraic manipulation and the other to explore the effects of filter pushdown.

5.2.1 Source Data—As source data, we created a set of RDF graphs similar to that in Table 1 but with varying number of triples. The way in which these graphs were generated was by determining a total number of individual subjects for each graph, then randomly deciding whether to create triples with a `:name`, `:email`, and/or `:phone` properties for each individual subject, and, if a triple is to be created, then randomly generating a string object. This resulted in graphs that permitted the obtention of different results for `LeftJoin` and `Join` operators. RDF graphs were generated for 10, 100, 1,000, and 10,000 individuals, resulting in the graphs containing 18, 175, 1,695, and 17,188 triples, respectively.

5.2.2 Measurement of Query Processing Cost—To measure the query processing cost, we calculated the total number of SPARQL-D decision operations performed by each graph pattern evaluation, that is, the total number of times in which a potential solution graph needed to be verified against a graph pattern. Given two solution sets Γ_1 and Γ_2 with sizes $|\Gamma_1|$ and $|\Gamma_2|$ and a value constraint r , we have the following:

1. `Filter( $r, \Gamma_1$ )` requires $|\Gamma_1|$ decisions.
2. `Join( $\Gamma_1, \Gamma_2$ )` requires $|\Gamma_1| * |\Gamma_2|$ decisions.
3. `Union( $\Gamma_1, \Gamma_2$ )` and `Or( $\Gamma_1, \Gamma_2$ )` require $|\Gamma_1| + |\Gamma_2|$ decisions.
4. `LeftJoin( $\Gamma_1, \Gamma_2, r$ )` requires $2 * (|\Gamma_1| * |\Gamma_2| + |\Gamma_1|)$ decisions.

For the case of i-d graph patterns, there is an additional processing cost needed for the disjunction correction algorithm.

Proposition 16: Let P be an SPARQL graph pattern, P' be its corresponding i-d graph pattern, $\Gamma(P, G)$ be the solution multiset for P , and $\Gamma'(P', G)$ be the solution set for P' against some RDF graph G . The SPARQL disjunction correction algorithm is linear with respect to the size of the multiset of solutions $\Gamma(P, G)$.

Proof: Let P' be the i-d graph pattern corresponding to graph pattern P , $\Gamma'(P', G)$ be the solution set for P' against G , and $\Gamma_u(P, G)$ be the partial solution multiset obtained after the resolution of the disjunction correction parse tree. Suppose that P contains q triple patterns. Then, every $\rho \in \Gamma_u(P, G)$ is the product of at most q operations. The elimination of noncompliant subgraphs in the last stage of the disjunction correction algorithm is basically a sorting problem. Finally, note that the size of the partial solution multiset $\Gamma_u(P, G)$ is linear with respect to the size of the SPARQL-conformant solution multiset $\Gamma(P, G)$, since the possible number of subgraphs for any graph ρ in $\Gamma(P, G)$ is given by the number of triples in ρ .

It is clear then that the number of decisions done for `Join` and `LeftJoin` operators are several orders of magnitude larger than for other operators or for the disjunction correction algorithm.

The number of decision operations for each graph pattern evaluation was plotted against the size of the RDF graph being queried, as measured in number of triples. For the case of our prototype implementation, this was measured directly; for comparison purposes, we used ARQ to determine the number of decisions at each SPARQL operator. This comparison should not

be understood as a comparison of our prototype against ARQ, since the latter contains optimization mechanisms that were not considered for our calculations. Rather, these comparisons show the ability to improve query processing efficiency through the algebraic manipulations and filter pushdown mechanisms proposed in this paper.

5.2.3 Algebraic Manipulation Test—The first test, whose results are illustrated in Fig. 2a, was designed to specifically evaluate the algebraic manipulation of i-d graph patterns. This test compares the SPARQL graph pattern illustrated in Fig. 1, labeled “Standard,” against two corresponding i-d graph patterns:

1. “semQA-1”: $\text{Or}(\text{Join}(P1, P2), P2, \text{Join}(P2, P3), \text{Join}(P1, P2, P3))$ and
2. “semQA-2”: $\text{Or}(\text{Join}(P1, \text{Or}(P1, P3)), \text{Join}(P2, \text{Or}(P2, P3)))$.

This test verified that the results from all three graph pattern evaluations were the same, further showing that the algorithms proposed in this paper result in SPARQL-compliant solutions. As can be observed, for all cases, the processing cost is linear with respect to the size of the source RDF graph, as expected from the data complexity analysis. These results also show the variation in processing cost resulting from the different algebraic expressions; this variation is highlighted in Fig. 2b, where only the results from the test on the RDF graph containing 1,695 triples is illustrated.

5.2.4 Filter Pushdown Test—The second test, whose results are illustrated in Fig. 3, was designed to evaluate the additional advantages of filter pushdown in the evaluation of i-d graph patterns. For this purpose, we modified the SPARQL graph pattern illustrated in Fig. 1 by adding a filter expression. This modified query is labeled “Std-withFilter” and was compared against two corresponding i-d graph patterns: one without filter pushdown applied (labeled “semQA-NoFP”) and the other with filter pushdown, labeled “semQA-FP.” Let $r_1 = \text{regex}(\text{name}; \text{"^ U([a-z][A-Z]) *"})$, $r_2 = \text{regex}(\text{phone}; \text{"^ 3[0-9]*"})$:

- a. “Std-with Filter”:
 - $\text{Filter}((r_1 \parallel r_2),$
 - LeftJoin
 - ♦ $(\text{Union}(\text{Join}(P1, P2), \text{LeftJoin}(P2, P3, \text{true})), \text{Join}(P2, P3), \text{true}))$
- b. “semQA-NoFP”:
 - $\text{Filter}((r_1 \parallel r_2), \text{Or}(\text{Join}(P1, P2), P2, \text{Join}(P2, P3), \text{Join}(P1, P2, P3)))$
- c. “semQA-FP”:
 - $\text{Or}(\text{Join}(\text{Filter}(r_1, P1), P2),$
 - $\text{Filter}(r_1, P1), P2,$
 - $\text{Join}(P2, \text{Filter}(r_2, P3)),$
 - $\text{Join}(\text{Filter}(r_1, P1), P2, P3),$
 - $\text{Join}(P1, P2, \text{Filter}(r_2, P3)))$.

As before, it was verified that in all three cases, the final SPARQL-conformant result multiset was the same. The results also illustrate again the linearity of processing cost with respect to the size of the RDF source graph. More significantly, it shows the substantial benefit in terms of processing cost that can be obtained from filter pushdown.

5.3 Experimental Evaluation of Complexity

To illustrate the complexity of processing of SPARQL queries, we used the RDF graph with 1,695 triples generated for the previous tests, and we devised a worst-case graph pattern evaluation. This permits us to find the upper bound of processing cost. As was stated in the previous section, the `Join` and `LeftJoin` operations are the most costly by orders of magnitude. Further, an operation `Join(P1, P2)` produces the largest number of results when `var(P1)` and `var(P2)` are disjoint, in which case the size of the result set is $|\text{Join}(P1, P2)| = |P1| * |P2|$. In order to conduct a worst-case graph pattern evaluation, we defined triple patterns $P_i = \{?x_i : \text{name } ?n_i\}$ and created graph patterns with varying numbers of triple patterns (and, therefore, varying number of variables) as $PV_n = \text{Join}(P1 \dots P_n)$, for $n = 2$ to 6 . We then proceeded to determine the total number of decision operations for each of these graph pattern evaluations and plotted the results against the number of triples n in each graph pattern. As illustrated in Fig. 4, it can be clearly seen that the worst-case evaluation of graph patterns is exponential in the number of terms in the patterns.

6 Conclusion

We present in this paper semQA, an extension to the SPARQL query algebra that uses an idempotent disjunction operator `Or`. semQA overcomes the limitations to algebraic manipulation imposed by the nondistributivity of `LeftJoin` over `Union`. A transformation of SPARQL graph patterns by substituting `Union` with `Or` is proposed. The resulting graph pattern, called an i-d graph pattern, can then be further operated upon through the use of algebraic equivalences. One particularly useful equivalence permits the replacement of the `LeftJoin` operator by a graph pattern containing `Or` and `Join` operators only. In addition, i-d graph patterns lend themselves better to filter pushdown, that is, to the algebraic manipulation of the query so that filters are evaluated early in query processing. In order to produce SPARQL-compliant results, an algorithm to derive the solution set of a graph pattern from the solution set of its corresponding i-d graph pattern is also presented. Additionally, we have presented an examination of the complexity of SPARQL, showing the language to be intractable in the general case. Finally, we present experimental results that validate the algorithms presented against the SPARQL standard, corroborate the complexity analysis, and illustrate the improvements in processing cost that can be achieved through the use of semQA algebraic manipulation and filter pushdown.

Future work includes three basic issues. First, we are completing the implementation of a complete SPARQL query engine using semQA. Second, we are working on the application of semQA to issues of data integration. In particular and in combination with algorithms for ontology alignment developed by our team [10], we expect to develop mechanisms to partition SPARQL queries into subqueries over different data sets, which can then be used to create distributed or federated SPARQL query engines. Finally, we are exploring approaches toward the design of improved algorithms for SPARQL query processing, based upon the similarity of the problem to the issue of subgraph isomorphism.

Acknowledgments

This research has been supported through Grant 1R43RR018667 from the National Center for Research Resources, US National Institutes of Health, and Grant 1R43CA132293 from the National Cancer Institute.

References

1. de Bruijn, J.; Tessaris, S.; Franconi, E. Logical Reconstruction of RDF and Ontology Languages; Proc Third Int'l Workshop Principles and Practice of Semantic Web Reasoning (PPSWR); 2005.

2. Codd EF. A Relational Model of Data for Large Shared Data Banks. *Comm ACM* 1970;13(6):377–387.
3. Cyganiak, R. HP Laboratories Bristol; 2005. A Relational Algebra for SPARQL. <http://www.hpl.hp.com/techreports/2005/HPL-2005-170.pdf>
4. Downey, R.; Fellows, M. *Parameterized Complexity*. Springer; 1999.
5. Downey, R.; Fellows, M.; Taylor, U. The Parameterized Complexity of Relational Database Queries and an Improved Characterization of W[1]. 2008. www.mcs.vuw.ac.nz/math/papers/DMTCS_96.ps
6. Flum, J.; Grohe, M. *Parameterized Complexity Theory*. Springer; 2006.
7. Frasnar F, Houben GJ, Vdovjak R, Barna P. RAL: An Algebra for Querying RDF. *World Wide Web* 2004;7(1):83–109.
8. Grohe M. Parameterized Complexity for the Database Theorist. *ACM SIGMOD Record* 2002;31(4): 86–96.
9. Huang X, Lai J, Jennings S. Maximum Common Subgraph: Some Upper Bound and Lower Bound Results. *BMC Bioinformatics* 2006;7(Suppl 4):S6. [PubMed: 17217524]
10. Jean-Mary, Y.; Kabuka, M. ASMOV: Ontology Alignment with Semantic Validation. *Proc. Joint Workshop Semantics, Ontologies, Databases (SWDB-ODDIS '07)*; 2007;
11. Manola, F.; Miller, E. *RDF Primer*, W3C recommendation. Feb. 2004 <http://www.w3.org/TR/rdf-primer/>
12. Papadimitriou C, Yannakakis M. On the Complexity of Database Queries. *J Computer and System Sciences* 1999;58(3):407–427.
13. Pérez, J.; Arenas, M.; Gutiérrez, C. Semantics and Complexity of SPARQL. *Proc Fifth Int'l Semantic Web Conf (ISWC '06)*. Nov. 2006 http://arxiv.org/PS_cache/cs/pdf/0605/0605124v1.pdf
14. Pérez, J.; Arenas, M.; Gutiérrez, C. Semantics of SPARQL. 2008. http://ing.utalca.cl/~jperez/papers/sparql_semantics.pdf
15. Polleres, A. SPARQL Rules!. GIA technical report, Universidad Rey Juan Carlos. Nov. 2006 <http://platon.escet.urjc.es/~axel/publications/GIA-TR-2006-11-28.pdf>
16. Prud'hommeaux, E.; Seaborne, A. SPARQL Query Language for RDF, W3C Recommendation. 2008. [updated 2008 Jan. 15, accessed 2008 Feb. 20], <http://www.w3.org/TR/rdf-sparql-query/>
17. Shironoshita EP, Ryan MT, Kabuka MR. Cardinality Estimation for the Optimization of Queries on Ontologies. *ACM SIGMOD Record*. June;2007
18. Vardi M. The Complexity of Relational Query Languages. *Proc 14th Ann ACM Symp Theory of Computing (STOC '82)* 1982:137–146.

Biographies

E. Patrick Shironoshita received the MS degree in electrical and computer engineering from the University of Miami, Coral Gables, Florida, in 1989. He is with INFOTECH Soft, Inc., where he is currently the project director for NIH-funded projects for the integration of information in genetic and medical research applications. His current research and development interests include semantic Web technologies, semantic query languages, and integration of genetics and medical information.

Yves R. Jean-Mary received the MS degree in computer engineering from the University of Miami in 2003. He is currently with INFOTECH Soft, Inc. His recent work includes the development of an ontology alignment algorithm called ASMOV. His research and development interests include ontology alignment and data integration.

Ray M. Bradley received the BS degree in computer engineering from the University of Miami in 2005. He is currently with INFOTECH Soft, Inc. His research and development interests include database integration and client-side Web technologies.

Mansur R. Kabuka received the PhD degree in computer engineering from the University of Virginia, Charlottesville, in 1983. He is a professor at the University of Miami, Florida, and the founder and president of INFOTECH Soft, Inc. His research interests include medical informatics, information processing, and medical imaging.

Fig. 1.
Example of parse tree processing for SPARQL disjunction-correction algorithm.

(a)

(b)

Fig. 2. Algebraic manipulation test. (a) Plot of processing cost versus RDF graph source size. (b) Comparison of processing cost for an RDF graph containing 1,695 triples.

Fig. 3.
Filter pushdown test.

Fig. 4.
Experimental evaluation of combined complexity.

TABLE 1

Example RDF Graph

Id	Triple	
t1	a:name	Alicia
t2	a:email	alicia@x.com
t3	a:phone	3056705111
t4	b:name	Benoit
t5	b:email	benoit@x.com

TABLE 2

Example SPARQL Queries

```
SELECT ?x ?n
WHERE { ?x :name ?n }
```

(a)

```
SELECT ?x ?n ?e
WHERE { ?x :name ?n . ?x :email ?e }
```

(b)

```
SELECT ?x ?n ?e
WHERE { ?x :name ?n . ?x :email ?e .
  FILTER (?n = "Alicia") }
```

(c)

```
SELECT ?x ?n ?e
WHERE {
  { ?x :name ?n }
  { ?x :email ?e }
}
```

(d)

```
SELECT ?x ?n ?p
WHERE {
  { ?x :name ?n } OPTIONAL
  { ?x :phone ?p }
}
```

(e)

```
SELECT ?x ?n ?p
WHERE {
  { ?x :name ?n } OPTIONAL
  { ?x :phone ?p . FILTER (?p = "3056705111") }
}
```

(f)

```
SELECT ?x ?n ?e
WHERE {
  { ?x :name ?n } UNION
  { ?x :email ?e }
}
```

(g)

(a) Simple query with single triple pattern. (b) Query with basic graph pattern. (c) Query with basic graph pattern with filter. (d) Query with group graph pattern. (e) Query with optional graph pattern. (f) Query with optional graph pattern with filter on the optional clause. (g) Query with alternative graph pattern.

TABLE 3

Example Evaluation of SPARQL Abstract Query Graph Patterns against RDF Graph G

```
eval(G, {?x :name ?n})
```

(a)

```
BGP({?x :name ?n . ?x :email ?e})
```

(b)

```
Filter((?n = "Alicia"),  
  BGP({?x :name ?n . ?x :email ?e}))
```

(c)

```
Join({?x :name ?n}, {?x :email ?e})
```

(d)

```
LeftJoin({?x :name ?n}, {?x :email ?e}, true)
```

(e)

```
LeftJoin({?x :name ?n}, {?x :email ?e},  
  (?p="3056705111"))
```

(f)

```
Union({?x :name ?n}, {?x :email ?e})
```

(g)

(a) Triple pattern. (b) Basic graph pattern. (c) Basic graph pattern with filter. (d) Group graph pattern. (e) Optional graph pattern. (f) Optional graph pattern with filter on optional clause. (g) Alternative graph pattern.

TABLE 4

Evaluation of LeftJoin and UNION Graph Patterns against the RDF Graph in Table 1

Id	Pattern
P1	?x :name ?n
P2	:a :name ?n
P3	?x :phone ?p

(a)

{ t1 . t2 }
{ t1 . t3 }
{ t4 . t5 }

(b)

{ t1 . t2 }
{ t1 . t3 }
{ t4 . t5 }
{ t4 }

(c)

(a) *Triple patterns.* (b) $P = \text{LeftJoin}(P1, \text{Union}(P2, P3), \text{true})$.
(c) $P' = \text{Union}(\text{LeftJoin}(P1, P2, \text{true}), \text{LeftJoin}(P1, P3, \text{true}))$.

TABLE 5

Example Evaluation of the Or Operator

Id	Pattern
P1	?x :name ?n
P2	?x :email ?e
P3	?x :phone ?p

(a)

{ t1 }

(b)

{t1 . t3}

(c)

(a) Triple patterns. (b) $P = \text{Or}(P1, P2)$. (c) $P' = \text{Or}((P1, \text{Join}(P2, P3)))$.

TABLE 6

SPARQL Disjunction Correction Algorithm Pseudocode

```

disjunction-correction( $\Gamma'$ , P) :
  Input set of i-d solution graphs  $\Gamma'$ ,
    SPARQL graph pattern P
   $\Gamma = \emptyset$ 
  For every  $\rho' \in \Gamma'$ 
     $\Gamma = \Gamma \cup \text{rebuild}(\rho', P)$ 
  Sort  $\Gamma$ 
  For every  $\rho \in \Gamma$ 
    If some  $\rho_x$  exists in  $\Gamma$  such that  $\rho \subset \rho_x$ 
      If  $\rho \setminus \rho_x$  is the solution for the right
        hand side of a LeftJoin operator
         $\Gamma = \Gamma \setminus \{\rho\}$ 
  Return  $\Gamma$ 

rebuild( $\rho'$ , P) :
  Input i-d solution graph  $\rho'$ ,
    SPARQL graph pattern P
  If P is a triple pattern
    Return  $\text{tx} \in \rho$  matching with P
  ElseIf P = Filter(r, P1)
    Return  $\{\rho \mid \rho \in \text{rebuild}(\rho', P1) \text{ and } r(\rho)\}$ 
  ElseIf P = Join(P1, P2)
    # NOTE:  $\times$  is the set cross-product operator
    Return  $\text{rebuild}(\rho', P1) \times \text{rebuild}(\rho', P2)$ 
  ElseIf P = LeftJoin(P1, P2)
    If  $\text{rebuild}(\rho', P2) = \emptyset$ 
      Return  $\text{rebuild}(\rho', P1)$ 
    Else
      Return  $\text{rebuild}(\rho', P1) \times \text{rebuild}(\rho', P2)$ 
  ElseIf P = Union(P1, P2)
    Return  $\text{rebuild}(\rho', P1) \cup \text{rebuild}(\rho', P2)$ 

```

TABLE 7

List of SPARQL Test Cases Tested

date-1	normalization-1	open-eq-6
date-2	normalization-2	open-eq-7
date-3	normalization-3	open-eq-8
date-4	open-cmp-1	open-eq-9
expr-1	open-cmp-2	open-eq-10
expr-2	open-eq-1	open-eq-11
expr-3	open-eq-2	open-eq-12
expr-4	open-eq-3	q-op-1
expr-5	open-eq-4	q-op-2
kanji-1	open-eq-5	q-op-complex-1
kanji-2		