Specification for tSPARQL

Olaf Hartig

December 25, 2008

Abstract

This document specifies tSPARQL, a trust-aware query language for trust weighted RDF graphs. A trust weighted RDF graph is basically an RDF graph where every triple is associated with a trust value. tSPARQL is a trust-aware extension to SPARQL, a query language for RDF. This document requires an understanding of the SPARQL query language and its semantics.

tSPARQL extends SPARQL in the following two ways. First, it adapts the SPARQL semantics to process the trust values associated with triples in the queried RDF graphs during query evaluation; i.e., it associates the solutions for graph patterns with trust values. Second, tSPARQL extends the query language by novel concepts to utilize the trust values in queries; i.e., it enables users to describe trust requirements and access the trust values associated to the solutions.

This document is structered as follows. First, Section 1 motivates the need for tSPARQL and gives an informal overview of its features. The remainder defines the tSPARQL query language in detail. It presents the tSPARQL approach for trust-aware processing of SPARQL queries (Section 2); followed by the tSPARQL extensions that enable the new tSPARQL features (Section 3). Finally, Section 4 concludes this document.

Contents

1 Introduction

Since its introduction the Resource Description Framework (RDF) [KC04] has become a well established data model for information about resources. Nowadays, it is widely accepted as the format for data on the Web. During recent years a large amount of data described by RDF has been published on the Web. For instance, the Semantic Web search engine Swoogle [DFJ+04] indexed about 2.3 million RDF documents in December 2007. The developers of Sindice [TOD07], a lookup index for Semantic Web documents, even reported more than 20 million documents in a recent blog post[1]. These news only indicate the beginning; content providers such as the review and rating service revyu.com[2] or the social tagging service GroupMe[3] publish their data in RDF, social network sites such as LiveJournal[4] provide FOAF[5] files for their user data, and legacy databases can easily be accessed as RDF datasources [BC06]. Another driving force for growing the „Web of data" is the Linking Open Data project[6] which creates and publishes RDF representations of various open datasets such as the Wikipedia [ABL+07] and the UniProt[7] proteine database. Besides publishing these datasets the goal of the Linking Open Data project is to interlink the resources described in them. In October 2007 the project offered 28 different datasets with about 3 million RDF links between them.

However, the openness of the Web which allows everyone to publish anything creates new challenges to applications and information consumers. Questions about reliability and trustworthiness are raised and become more important. Doubtful sources may provide questionable or unjustified information; malicious users may publish misleading or wrong information. Even some kind of RDF spam may emerge, i.e. irrelevant, futile or manipulative statements. Such unreliable data could dominate the result of queries, taint inferred data, affect local knowledge bases, and have negative or misleading impact on software agents. These issues must be addressed. Current approaches such as Eigentrust [KSGM03], PeerTrust [XL04], and Appleseed [ZL04] propose (parts of) trust infrastructures based on a Web of Trust. Even if some of the proposals consider the reliability of the data provided by members of the network the majority focuses on the trustworthiness of the members.

What is missing for the Web of data is a uniform way to rate the trustworthiness of the information on the Web and standardized mechanisms to access and to use the ratings. To rate trustworthiness in the Web of data we propose a trust model for RDF data that represents the trustworthiness of every triple by a trust value. To specify the trust value associated with a triple we introduce a *trust function*.

Definition 1.1 Let T be the set of all RDF triples. A **trust function** tv^C for RDF triples is a mapping

$$\mathrm{tv}^C : T \rightarrow \{tv \mid tv \in [\text{-}1, 1]\} \cup \{\emptyset\}$$

[1] http://blog.sindice.com/?p=5
[2] http://revyu.com
[3] http://groupme.org
[4] http://www.livejournal.com
[5] http://www.foaf-project.org
[6] http://esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData
[7] http://beta.uniprot.org/

which assigns every triple in T a subjective trust value that represents the trustworthiness of the triple specific to an information consumer C. □

The trust value $tv^C(t)$ for the triple t defines the degree of belief of an information consumer C in the truth of the fact stated by triple t. The value 1 represents absolute belief; -1 represents absolute disbelief; 0 represents the lack of belief/disbelief. Furthermore, we permit unknown trust values, denoted by $\emptyset$, for cases where it is impossible to determine the trustworthiness of triples.

We refer to an RDF graph where all triples that are associated with trust values for a specific information consumer as a *trust weighted RDF graph*.

Definition 1.2 A **trust weighted RDF graph** $\widetilde{G}^C$ for information consumer C is a pair (G, tv^C) consisting of an RDF graph G and a trust function tv^C. □

Example 1.1 *Figure 1 depicts a trust weighted RDF graph. The edges represent the predicates of triples. They are annotated with the predicate identifier as usual [KC04] and with an additional label for the trust value of the corresponding triple. The sample graph consists of three triples. One of them asserts that resource* `ex:Alice` *is a person and is associated with a trust value of* 0.8. □

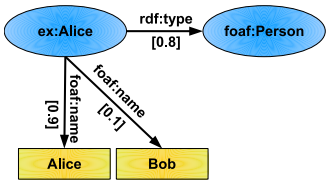

Figure 1: A trust weighted RDF graph

Users as well as software agents have to be able to utilize the trust values in trust weighted RDF graph and base their decisions upon these values. They have to be enabled to ask queries such as:

- Q_1 Return the title of all courses attended by Alice where I can highly trust information on the attendance and the correct title.
- Q_2 Return the title of all courses attended by Alice where I cannot, at least moderately, trust in the single fact she really attended.
- Q_3 Return the name of the lecturer who gives a specific course; additionally, return the trustworthiness of is this information.
- Q_4 Return all courses attended by a student ordered by the trustworthiness of is this information.
- Q_5 For all persons living in the same city as me, return their interests together with the trustworthiness of the fact they are interested in the topic returned.
- Q_6 Return a list of all people with two different name properties where the trustworthiness of both differ by at least 0.3.

These sample queries are of two different types regarding the way they access trust values. Q_1 and Q_2 describe requirements regarding the trustworthiness of query results or parts of them; i.e., they filter results with respect to the

trust values of the solutions. In contrast, Q_3 to Q_6 use trust values in the query. Fundamental to enable information consumers to ask queries such as Q_1 to Q_6 is the possibility to embed declarative descriptions of trustworthiness requirements into queries. Hence, we propose the trust-aware query language tSPARQL. Based on our trust model tSPARQL is an extension to the query language SPARQL [PS08] for RDF.

To access trust values tSPARQL contains the `TRUST AS` clause. Consider the query in Figure 2 which asks for names of students and their courses. It contains a `TRUST AS` clause with a new variable `?t`. `?t` allows access to the trust value associated with the triples that match the pattern in line 7. Hence, the query additionally asks for the trustworthiness of the fact that the student really takes the respective course.

```
1 PREFIX ub: <http://www.lehigh.edu/.../univ-bench.owl#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 SELECT ?n ?c ?t
4 WHERE {
5     { ?s rdf:type ub:Student .
6       ?s ub:name ?n }
7     { ?s ub:takesCourse ?c .
8       TRUST AS ?t }
9 }
```

Figure 2: Example query with trust projection

The `TRUST AS` clause offers the following novel features: i) the new variable may become part of the query result, ii) the variable may be used for sorting the results, iii) it may be associated with parts of the query pattern, and two variables that represent trust values of different query pattern parts can be compared. Hence, `TRUST AS` enables users to express queries Q_3 to Q_6. Besides accessing trust values a `TRUST AS` clause can even be used to express trust requirements as in queries Q_1 and Q_2; in addition to the `TRUST AS` clause users simply add a `FILTER` clause which restricts the new variable. However, for convenience tSPARQL contains another new clause for these cases, namely the `ENSURE TRUST` clause. Figure 3 depicts a query with an `ENSURE TRUST` clause. This clause comprises a pair of numbers in brackets, where the first number denotes a lower bound and the second number denotes an upper bound. Again, the query asks for names of students and their courses; however, in this case only those solutions become part of the result where we highly trust that the student really takes the respective course.

```
1 PREFIX ub: <http://www.lehigh.edu/.../univ-bench.owl#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 SELECT ?n ?c
4 WHERE {
5     { ?s rdf:type ub:Student .
6       ?s ub:name ?n }
7     { ?s ub:takesCourse ?c .
8       ENSURE TRUST (0.9,1.0) }
9 }
```

Figure 3: Example query with ensure trust constraint

To enable the extensions tSPARQL basically enhances SPARQL in two ways. First, it extends the query language with the new clauses and defines operations for the clauses. Second, tSPARQL extends the processing of queries to consider trust values because trust values are currently not part of SPARQL.

2 Trust in SPARQL Query Processsing

The semantics of SPARQL [PS08] do not consider trust values. To implement trust-aware query processing tSPARQL adapts parts of the semantics, especially the corresponding concepts of query evaluation. In this section, we first take a closer look at SPARQL query processing and query evaluation; we then present our extensions.

The SPARQL specification gives an operational definition of the semantics of SPARQL. In brief, the specification defines a grammar for the query language, a translation from a parse tree to an abstract syntax tree (AST), a transformation from an AST to an abstract query with an algebra expression, and an operation to evaluate abstract queries based on algebra operators. The algebra is defined to calculate query solutions and operate on them (e.g. merge solutions from different parts of the query). Finally, a result form definition specifies how to create the query result from the solutions.

To consider trust values during query processing and to enable algebra operators that access trust values tSPARQL must extend query evaluation. E.g., the trust values have to become part of the solutions. Hence, tSPARQL redefines the notion of solutions. Additionally, tSPARQL specifies how these solutions are calulated and how the algebra operates on them. The following sections address these issues.

2.1 Trust-aware Basic Graph Pattern Matching

This section defines the notion of solutions in the context of tSPARQL. It follows the definitions of *solution mapping* and *solution*s for BGPs as defined in the SPARQL specification. However, solutions in tSPARQL must contain a trust value. Therefore, the basic idea is to associate solution mappings with trust values. In the following we refer to them as *trust weighted solution mappings*.

Definition 2.1 A **trust weighted solution mapping** $\widetilde{\mu}$ is a pair (μ, t) consisting of a solution mapping μ (as defined in [PS08, Section 12.1.6]) and a trust value t with $-1 \leq t \leq 1$ or $t = \emptyset$. □

Every solution mapping that is a solution to a BGP represents one matching subgraph; the trust value of this solution mapping represents the trustworthiness of the subgraph. The trustworthiness of a subgraph is an aggregation of the trustworthiness of its triples which is represented by trust values that are specified by a trust function (cf. Definition 1.1). Hence, the trustworthiness of the subgraph can be represented by a trust value that is calculated from the trust values of its triple using a *trust aggregation function*.

Definition 2.2 A **trust aggregation function** ta for trust weighted RDF graphs is a function that assigns an aggregated trust value $ta(\widetilde{G^C})$ from the set $\{t \mid -1 \leq t \leq 1\} \cup \{\emptyset\}$ to a trust weighted RDF graph $\widetilde{G^C}$. □

Following the definition from the SPARQL specification, we define *solutions for a BGP* over a trust weighted RDF graph.

Definition 2.3 Let b be a basic graph pattern; let $\widetilde{G^C} = (G, tv^C)$ be a trust weighted RDF graph for information consumer C. A trust weighted solution mapping (μ, t) is a **solution** for b in $\widetilde{G^C}$ if there is a pattern instance mapping P (as defined in [PS08, Section 12.3.1]) such that $P(b)$ is a subgraph of G, μ is the restriction of P to the query variables in b, and t is the aggregated trust value

$$t = ta\left(\widetilde{P(b)^C}\right)$$

of the trust weighted RDF graph $\widetilde{P(b)^C} = (P(b), tv^C)$ calculated by a trust aggregation function ta. For each solution μ for b $\text{card}_{\widetilde{\Omega}}(\widetilde{\mu})$ is the number of distinct pattern instance mappings $P = \mu(\sigma)$ such that $P(b)$ is a subgraph of G. □

With the given definition of solution the result of BGP matching over a trust weighted RDF graph is a multiset of trust weighted solution mappings; the solution mapping of a trust weighted solution mapping can be part of different pattern instance mappings that represent different matching subgraphs and the trust value for some of them can be the same.

Notice, our definition does not prescribe a specific trust aggregation function. Applications have the freedom to choose an aggregation function that fits their use cases. Possible trust aggregation functions include

$$ta_{\min}\left(G, tv^C\right) = \begin{cases} \emptyset & \text{if } \exists t \in G : tv^C(t) = \emptyset \\ \min\{tv^C(t) | t \in G\} & \text{else} \end{cases}$$

and

$$ta_{\text{avg}}\left(G, tv^C\right) = \begin{cases} \emptyset & \text{if } \exists t \in G : tv^C(t) = \emptyset \\ \frac{1}{\|G\|} \sum_{t \in G} tv^C(t) & \text{else} \end{cases}$$

Example 2.1 *When we apply the BGP in Figure 4(a) to the sample trust weighted RDF graph in Figure 1 we find two matching subgraphs resulting in the two solution mappings shown in Figure 4(b).* μ_1 *maps* `?s` *to* `ex:Alice` *and* `?n` *to the literal* Alice*;* μ_2 *maps* `?s` *to* `ex:Alice` *again and* `?n` *to the string* Bob*. To determine the trust values for both,* μ_1 *and* μ_2*, we choose* $ta_{\min}$ *as our application-specific trust aggregation function. This is a reasonable choice if we assume the solution of a BGP is only as trustworthy as the least trusted triple in the matching subgraph. The subgraph for* μ_1 *consists of two triples with trust values 0.8 and 0.9, respectively. Hence, our first solution is the trust weighted solution mapping* $\widetilde{\mu_1} = (\mu_1, 0.8)$*. For* μ_2 *we have the two trust values 0.8 and 0.1; our second solution is* $\widetilde{\mu_2} = (\mu_2, 0.1)$*.* □

Notice, conceptually BGP matching is entirely independent from the applied method to determine the trust values of the matching subgraphs. By not prescribing a specific trust function tv^C for the queried trust weighted RDF graph we do not prescribe a method to determine the trust values. The clear separation of the two tasks, determining trust values and BGP matching, is an advantage of our approach. Nonetheless, in practice it may be beneficial to combine the implementation of both tasks. For instance, if determining the trust values for

?s rdf:type foaf:Person .
?s foaf:name ?n .

(a)

μ		t
?s	?n	
ex:Alice	"Alice"	0.9
ex:Bob	"Bob"	0.1

(b)

Figure 4: A BGP (a) and its solutions (b) for the trust weighted RDF graph in Figure 1

a set of triples from the same source is more efficient than considering every triple on its own an algorithm for trust-aware BGP matching might be adjusted accordingly to become more efficient. Another example is the caching of trust values determined for the triples in a matching subgraph and the employment of these caches in subsequent BGP matchings.

In order to operate on trust weighted solution mappings tSPARQL adjusts the algebra operators from SPARQL. Some of the operators merge two solutions and the SPARQL specification defines the merging of solution mappings. However, the adjusted operators additionally have to consider the trust values while merging. The trust value of a merged solution mapping is an aggregation of the trust values associated with the individual mappings that are being merged. For this purpose tSPARQL introduces another type of aggregation functions, called *trust merge function*.

Definition 2.4 A **trust merge function** tm for two trust weighted solution mappings $\widetilde{\mu_1}$ and $\widetilde{\mu_2}$ is a commutative and associative function that determines a merged trust value $tm(\widetilde{\mu_1}, \widetilde{\mu_2})$ from the set $\{t \mid -1 \leq t \leq 1\} \cup \{\emptyset\}$. □

Possible trust merge functions are

$$tm_{\min}(\widetilde{\mu_1}, \widetilde{\mu_2}) = \begin{cases} \emptyset & \text{if } t_1 = \emptyset \vee t_2 = \emptyset \\ \min(t_1, t_2) & \text{else} \end{cases}$$

and

$$tm_{\text{avg}}(\widetilde{\mu_1}, \widetilde{\mu_2}) = \begin{cases} \emptyset & \text{if } t_1 = \emptyset \vee t_2 = \emptyset \\ \frac{1}{2}(t_1 + t_2) & \text{else} \end{cases}$$

where $\widetilde{\mu_i} = (\mu_i, t_i)$.

2.2 Trust-aware Algebra for SPARQL

After defining trust weighted solution mappings this section explains how these mappings are combined in more complex queries. Besides BGPs, the SPARQL specification introduces other graph patterns. During query evaluation they are represented by algebra operators which operate on multisets of solution mappings. For the new clauses (cf. Section 1) tSPARQL needs new types of operators. To enable these new operators to access the trust values in solutions all algebra operators have to consider the trust values. Hence, tSPARQL redefines the conventional SPARQL algebra operators to operate on multisets of trust weighted solution mappings.

For a precise redefinition of the algebra operators we introduce the following symbols (following the corresponding symbols in [PS08]). With $\text{card}_{\widetilde{\Omega}}(\widetilde{\mu})$ we

denote the cardinality of the trust weighted solution mapping $\widetilde{\mu}$ in a multiset $\widetilde{\Omega}$ of trust weighted solution mappings; with $\text{card}_{\widetilde{\Psi}}(\widetilde{\mu})$ we denote the cardinality of the trust weighted solution mapping $\widetilde{\mu}$ in a sequence $\widetilde{\Psi}$ of trust weighted solution mappings.

tSPARQL redefines the join operator as follows.

Definition 2.5 Let $\widetilde{\Omega_1}$ and $\widetilde{\Omega_2}$ be multisets of trust weighted solution mappings. The result of a **join operator** is a multiset of trust weighted solution mappings which is defined as

$$Join\left(\widetilde{\Omega_1},\widetilde{\Omega_2}\right) = \left\{ \left(merge(\mu_1,\mu_2), tm\left(\widetilde{\mu_1},\widetilde{\mu_2}\right)\right) \mid \widetilde{\mu_1} = (\mu_1,t_1) \in \widetilde{\Omega_1} \wedge \widetilde{\mu_2} = (\mu_2,t_2) \in \widetilde{\Omega_2} \wedge \mu_1 \text{ and } \mu_2 \text{ are compatible} \right\}$$

with

$$\text{card}_{Join\left(\widetilde{\Omega_1},\widetilde{\Omega_2}\right)}(\widetilde{\mu}) = \sum_{\substack{\widetilde{\mu_1}\in\widetilde{\Omega_1}\\ \widetilde{\mu_2}\in\widetilde{\Omega_2}}} \begin{cases} \text{card}_{\widetilde{\Omega_1}}(\widetilde{\mu_1}) \cdot \text{card}_{\widetilde{\Omega_2}}(\widetilde{\mu_2}) & \text{if } \widetilde{\mu} = (\mu,t) \text{ with } t = tm\left(\widetilde{\mu_1},\widetilde{\mu_2}\right) \text{ and } \mu = merge(\mu_1,\mu_2) \text{ where } \widetilde{\mu_i} = (\mu_i,t_i) \\ 0 & \text{else} \end{cases}$$

where *merge* is the merge operation for solution mappings [PS08, Section 12.3] and *tm* is an application-specific trust merge function. □

Notice, the definition does not prescribe a specific trust merge function; thus giving applications a choice. The following example illustrates trust-aware query evaluation.

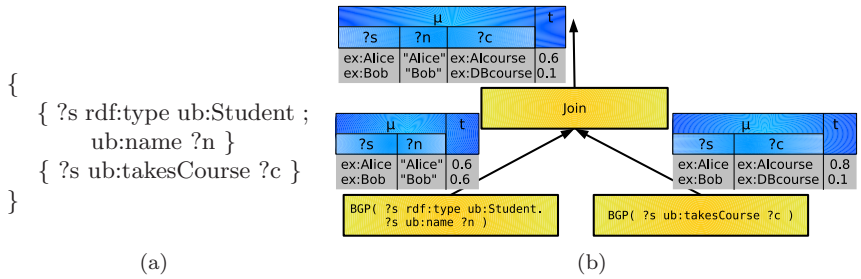

Figure 5: Group graph pattern (a) and the representing operator tree with solutions (b)

Example 2.2 *The group graph pattern in Figure 5(a) groups two BGPs that ask for the names of students and their courses. Figure 5(b) depicts the corresponding algebra expression as an operator tree annotated with trust weighted solution mappings that represent some sample solutions. Consider the two sample solution sets from both BGP matchings represented by the two tables near the bottom of Figure 5(b). Joining the solutions from both multisets results in the solutions represented by the upper table. The trust merge function applied is* $tm_{\min}$*, i.e. for every merged solution pair we select the lower trust value for the resulting solution. This is reasonable if we assume a merged solution mapping is only as trustworthy as the least trusted mapping used for merging.* □

Similiar to the join operator tSPARQL redefines all operators of the SPARQL algebra. What follows is a list of the definitions.

Definition 2.6 Let $\widetilde{\Omega}$ be a multiset of trust weighted solution mappings; let ex be an expression as defined in [PS08]. The result of a **filter operator** is a multiset of trust weighted solution mappings which is defined as

$$Filter\left(ex, \widetilde{\Omega}\right) = \Big\{\widetilde{\mu} \mid \widetilde{\mu} = (\mu, t) \in \widetilde{\Omega} \wedge ex(\mu) \text{ is an expression that has an effective boolean value of true}\Big\}$$

with

$$\text{card}_{Filter\left(ex,\widetilde{\Omega}\right)}(\widetilde{\mu}) = \text{card}_{\widetilde{\Omega}}(\widetilde{\mu})$$ □

Definition 2.7 Let $\widetilde{\Omega_1}$ and $\widetilde{\Omega_2}$ be multisets of trust weighted solution mappings; let ex be an expression as defined in [PS08]. The result of a **diff operator** is a multiset of trust weighted solution mappings which is defined as

$$\begin{aligned} Diff\left(\widetilde{\Omega_1}, \widetilde{\Omega_2}, ex\right) = \quad & \Big\{\widetilde{\mu_1} \,\Big|\, \widetilde{\mu_1} = (\mu_1, t_1) \in \widetilde{\Omega_1} \wedge \\ & \quad \text{for all } \widetilde{\mu_2} = (\mu_2, t_2) \in \widetilde{\Omega_2} \text{ holds:} \\ & \quad \mu_1 \text{ and } \mu_2 \text{ are not compatible}\Big\} \cup \\ & \Big\{\widetilde{\mu_1} \,\Big|\, \widetilde{\mu_1} = (\mu_1, t_1) \in \widetilde{\Omega_1} \wedge \\ & \quad \text{for all } \widetilde{\mu_2} = (\mu_2, t_2) \in \widetilde{\Omega_2} \text{ where } \mu_1 \text{ and } \mu_2 \text{ are compatible it holds:} \\ & \quad ex\Big(merge\big(\mu_1, \mu_2\big)\Big) \text{ has an effective boolean value of false}\Big\} \end{aligned}$$

with

$$\text{card}_{Diff\left(\widetilde{\Omega_1},\widetilde{\Omega_2},ex\right)}(\widetilde{\mu}) = \text{card}_{\widetilde{\Omega_1}}(\widetilde{\mu})$$

where $merge$ is the merge operation for solution mappings [PS08]. □

Definition 2.8 Let $\widetilde{\Omega_1}$ and $\widetilde{\Omega_2}$ be multisets of trust weighted solution mappings; let ex be an expression as defined in [PS08]. The result of a **left join operator** is a multiset of trust weighted solution mappings which is defined as

$$LJoin\left(\widetilde{\Omega_1}, \widetilde{\Omega_2}, ex\right) = Filter\Big(ex, Join\big(\widetilde{\Omega_1}, \widetilde{\Omega_2}\big)\Big) \cup Diff\left(\widetilde{\Omega_1}, \widetilde{\Omega_2}, ex\right)$$

with

$$\text{card}_{LJoin\left(\widetilde{\Omega_1},\widetilde{\Omega_2},ex\right)}(\widetilde{\mu}) = \text{card}_{Filter\left(ex,Join\left(\widetilde{\Omega_1},\widetilde{\Omega_2}\right)\right)}(\widetilde{\mu}) + \text{card}_{Diff\left(\widetilde{\Omega_1},\widetilde{\Omega_2},ex\right)}(\widetilde{\mu})$$

□

Definition 2.9 Let $\widetilde{\Omega_1}$ and $\widetilde{\Omega_2}$ be multisets of trust weighted solution mappings. The result of a **union operator** is a multiset of trust weighted solution mappings which is defined as

$$Union\left(\widetilde{\Omega_1}, \widetilde{\Omega_2}\right) = \widetilde{\Omega_1} \cup \widetilde{\Omega_2}$$

with

$$\text{card}_{Union\left(\widetilde{\Omega_1},\widetilde{\Omega_2}\right)}(\widetilde{\mu}) = \text{card}_{\widetilde{\Omega_1}}(\widetilde{\mu}) + \text{card}_{\widetilde{\Omega_2}}(\widetilde{\mu})$$ □

Definition 2.10 Let $\widetilde{\Omega}$ be a multiset of trust weighted solution mappings. The result of a **to-list operator** is a sequence of trust weighted solution mappings which is defined as

$$ToList\left(\widetilde{\Omega}\right) = \left[\widetilde{\mu} \mid \widetilde{\mu} \in \widetilde{\Omega}\right]$$

with

$$\mathrm{card}_{ToList(\widetilde{\Omega})}(\widetilde{\mu}) = \mathrm{card}_{\widetilde{\Omega}}(\widetilde{\mu})$$

□

Definition 2.11 Let $\widetilde{\Psi}$ be a sequence of trust weighted solution mappings; let *cond* be a condition as defined in [PS08]. The result of an **order-by operator** is a sequence of trust weighted solution mappings which is defined as

$$OrderBy\left(\widetilde{\Psi}, cond\right) = \left[\widetilde{\mu} \mid \widetilde{\mu} \in \widetilde{\Psi} \wedge \text{the sequence satisfies } cond\right]$$

with

$$\mathrm{card}_{OrderBy(\widetilde{\Psi},cond)}(\widetilde{\mu}) = \mathrm{card}_{\widetilde{\Psi}}(\widetilde{\mu})$$

□

Definition 2.12 Let $\widetilde{\Psi}$ be a sequence of trust weighted solution mappings; let PV be a set of query variables as defined in [PS08]. The result of a **project operator** is a sequence of trust weighted solution mappings which is defined as

$$Project\left(\widetilde{\Psi}, PV\right) = \left[\left(proj\left(\mu, PV\right), t\right) \mid \widetilde{\mu} = (\mu, t) \in \widetilde{\Psi}\right]$$

with

$$\mathrm{card}_{Project(\widetilde{\Psi},PV)}(\widetilde{\mu}) = \mathrm{card}_{\widetilde{\Psi}}(\widetilde{\mu})$$

where $proj\left(\mu, PV\right)$ is a mapping from a solution mapping μ to a solution mapping that is restricted to the variables in PV. The order of $Project\left(\widetilde{\Psi}, PV\right)$ must preserve any ordering given by $OrderBy$. □

Definition 2.13 Let $\widetilde{\Psi}$ be a sequence of trust weighted solution mappings. The result of a **distinct operator** is a sequence of trust weighted solution mappings which is defined as

$$Distinct\left(\widetilde{\Psi}\right) = \left[\widetilde{\mu} \mid \widetilde{\mu} \in \widetilde{\Psi}\right]$$

with

$$\mathrm{card}_{Distinct(\widetilde{\Psi})}(\widetilde{\mu}) = 1$$

The order of $Distinct\left(\widetilde{\Psi}\right)$ must preserve any ordering given by $OrderBy$. □

Definition 2.14 Let $\widetilde{\Psi}$ be a sequence of trust weighted solution mappings. The result of a **distinct operator** is a sequence of trust weighted solution mappings which is defined as

$$Reduced\left(\widetilde{\Psi}\right) = \left[\widetilde{\mu} \mid \widetilde{\mu} \in \widetilde{\Psi}\right]$$

with

$$1 \leq \mathrm{card}_{Reduced(\widetilde{\Psi})}(\widetilde{\mu}) \leq \mathrm{card}_{\widetilde{\Psi}}(\widetilde{\mu})$$

The order of $Reduced\left(\widetilde{\Psi}\right)$ must preserve any ordering given by $OrderBy$. □

Definition 2.15 Let $\widetilde{\Psi}$ be a sequence of trust weighted solution mappings; let s and l be natural numbers. The result of a **distinct operator** is a sequence of trust weighted solution mappings which is defined as

$$Slice\left(\widetilde{\Psi}, s, l\right) = \Big[\widetilde{\mu} \mid \widetilde{\mu} \in \widetilde{\Psi} \wedge \text{ the position of } \widetilde{\mu} \text{ in } \widetilde{\Psi} \text{ is in the interval } [s, s+l-1]\Big]$$

The order of $Slice(\widetilde{\Psi})$ must preserve any ordering given by $OrderBy$. □

3 SPARQL Extension for Trust Requirements

Section 1 gives a high-level overview of the use of trust values in tSPARQL queries. This section provides a more formal description of the extension to the SPARQL query language, i.e. the `TRUST AS` and `ENSURE TRUST` clauses. To enable the new clauses tSPARQL

- extends the grammar of SPARQL,
- modifies the translation from an abstract syntax tree to an algebra expression,
- defines new algebra operators,
- and extends the evaluation semantics.

The remainder of this section covers these topics in the given order.

3.1 tSPARQL Grammar

On top of the SPARQL query language tSPARQL introduces two new clauses, namely the `TRUST AS` and the `ENSURE TRUST` clause. Both new clauses can occur at any position in a query where `FILTER` clauses are permitted. The `TRUST AS` clause is denoted by the keywords `TRUST AS` which are followed by a query variable. This variable must not be contained in any other pattern of the query. The `ENSURE TRUST` clause is denoted by the keywords `ENSURE TRUST` which is followed by two real numbers in brackets; the first number represents the lower bound and the second number is the upper bound. An excerpt of the extended SPARQL syntax is listed in Figure 6. The listing shows the relevant symbols that have been adjusted and added for tSPARQL; the bold faced parts represent the tSPARQL-specific additions.

3.2 Converting Graph Patterns

Based on the SPARQL grammar the SPARQL specification defines "the process of converting graph patterns and solution modifiers in a SPARQL query string into a SPARQL algebra expression" [PS08, Section 12.2]. To consider the extended grammar for tSPARQL (cf. Section 3.1) the process must be adjusted accordingly. The new clauses are part of the query graph pattern (i.e. the WHERE clause). Hence, tSPARQL has to adjust the translation of graph patterns to algebra expressions which is defined in a functional manner

GroupGraphPattern ::= '{' TriplesBlock?
((GraphPatternNotTriples | Filter | **EnTrust** | **PrTrust**)
'.'? TriplesBlock?
)*
'}'

EnTrust ::= 'ENSURE' 'TRUST' '(' NumericLiteral ',' NumericLiteral ')'

PrTrust ::= 'TRUST' 'AS' Var

Var ::= VAR1 | VAR2

NumericLiteral ::= NumericLiteralUnsigned | NumericLiteralPositive | NumericLiteralNegative

NumericLiteralUnsigned ::= INTEGER | DECIMAL | DOUBLE

NumericLiteralPositive ::= INTEGER_POSITIVE | DECIMAL_POSITIVE | DOUBLE_POSITIVE

NumericLiteralNegative ::= INTEGER_NEGATIVE | DECIMAL_NEGATIVE | DOUBLE_NEGATIVE

Figure 6: Extensions for tSPARQL to the SPARQL syntax

by specifying a recursive `Transform` procedure [PS08, Section 12.2.1]. The input to `Transform` is a graph pattern as defined by the grammar; the result is an algebra expression. Since there are different types of graph patterns the definition of `Transform` is subdivided. This document redefines the part of the definition which considers the graph patterns of the syntax form of a `GroupGraphPattern` because this is the only grammar symbol which has been extended for tSPARQL (cf. Figure 6). Figure 7 shows the adjusted part of the definition; the bold faced parts represent the tSPARQL-specific additions. The adjusted definition uses the following two new algebra symbols:

$$\mathrm{PT}(Pattern, Variable) \qquad \text{and} \qquad \mathrm{ET}(Pattern, Number, Number)$$

3.3 tSPARQL Algebra

To evaluate the new `TRUST AS` and `ENSURE TRUST` clauses tSPARQL needs proper algebra operators. This section defines the *project trust operator* and the *ensure trust operator*. As the redefined SPARQL algebra operators (cf. Section 2.2) the two new operators operate on a multiset of trust weighted solution mappings.

3.3.1 Project Trust Operator

The project trust operator evaluates the `TRUST AS` clause. For every mapping the operator accesses the trust value, creates a new variable binding which maps the specified variable to an RDF literal that represents the trust value, and adds the new binding to the mapping.

Definition 3.1 Let $\widetilde{\Omega}$ be a multiset of trust weighted solution mappings; let v be a query variable which is not bound in any $\widetilde{\mu} \in \widetilde{\Omega}$; let $\mathrm{L}(t)$ be a function that returns an RDF literal of type `xsd:float` with the value of t. The result of a **project trust operator** is a multiset of trust weighted solution mappings which is defined as

$$PT\left(v, \widetilde{\Omega}\right) = \left\{(\mu', t) \mid (\mu, t) \in \widetilde{\Omega} \wedge \mu' = \mu \cup \left\{(v, \mathrm{L}(t))\right\}\right\}$$

Let $FS := \emptyset$; /* *the empty set* */
Let $G :=$ the empty pattern; /* *a basic graph pattern which is the empty set* */
Let TV **:=** $\emptyset$**;**
Let TC **:=** $\emptyset$**;**

FOR EACH element E in the `GroupGraphPattern`
  IF E is of the form `FILTER(expr)`
    $FS := FS \cup \{$`expr`$\}$;
  IF E **is of the form** `TRUST AS v`
    TV **:=** $TV \cup \{$`v`$\}$**;**
  IF E **is of the form** `ENSURE TRUST (l,u)`
    TC **:=** $TC \cup \{($`l`$,$`u`$)\}$**;**
  IF E is of the form `OPTIONAL{P}`
  THEN
    Let $A :=$ Transform(`P`);
    IF A is of the form `Filter(F, A2)`
      $G :=$ LJoin(G, `A2`, `F`);
    ELSE
      $G :=$ LJoin(G, A, true);
  IF E is any other form:
    Let $A :=$ Transform(E);
    $G :=$ Join(G, A);

IF $TC \neq \emptyset$
  FOR EACH pair (l, u) **in** TC
    G **:= ET(**G**,** l**,** u**);**

IF $VC \neq \emptyset$
  FOR EACH variable v **in** TV
    G **:= PT(**G**,** v**);**

IF $FS \neq \emptyset$
  Let $X :=$ Conjunction of expressions in FS;
  $G :=$ Filter(X, G);

The result is G.

Figure 7: Adjusted `Transform` procedure to consider the tSPARQL grammar

with

$$\text{card}_{PT(v,\widetilde{\Omega})}(\widetilde{\mu}) = \text{card}_{\widetilde{\Omega}}(\widetilde{\mu}) \qquad \square$$

The following example illustrates query evaluation with a project trust operator.

Example 3.1 *Figure 8(a) depicts the operator tree, annotated with sample solutions, for the query pattern of the query in Figure 2. Compare the solutions consumed and provided by the project trust operator. Every solution provided by this operator contains an additional binding for variable* `?t`*. This binding maps* `?t` *to a value that corresponds to the trust value that is associated with the respective solution when the project trust operator is evaluated (e.g.* 0.8 *for the first solution). Note, we use the trust merge function* $tm_{\min}$ *for the join operation. Thus, the trust value of the first solution after the join is* 0.6*. However, the value bound to variable* `?t` *has not changed; it is still* 0.8*. This can be*

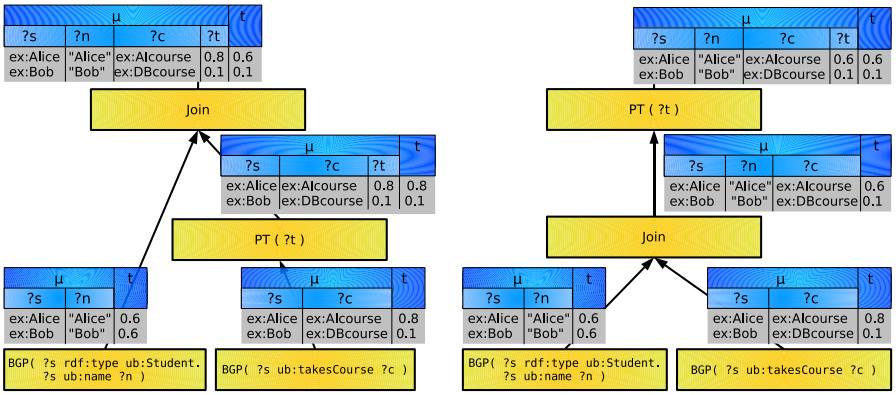

Figure 8: Project trust operators in an operator tree with sample solutions

attributed to the limited scope of the trust projection and reflects the intention of the `TRUST AS` *clause and its position in the query.*

To illustrate the role of the limited scope consider a slight variation of the query in Figure 2 where the `TRUST AS` *clause has been defined for the whole group graph pattern (i.e. before the last closing brace in line 8). Figure 8(b) depicts the corresponding operator tree annotated with sample solutions. Notice, the solutions from BGP matching are the same as in Figure 8(a). Even so, the first of the overall resulting solutions differ for* `?t` *because the project trust operator is applied after joining the solutions. Obviously, the position of a* `TRUST AS` *clause in a query pattern matters.* □

3.3.2 Ensure Trust Operator

The ensure trust operator evaluates the `ENSURE TRUST` clause. The operator accepts only these solutions that have a trust value within the specified interval, i.e., it eliminates any solutions with trust values lesser than the lower bound or larger than the upper bound.

Definition 3.2 Let $\widetilde{\Omega}$ be a multiset of trust weighted solution mappings; let $l, u \in [-1, 1]$ be lower and upper bound values, respectively. The result of an **ensure trust operator** is a multiset of trust weighted solution mappings which is defined as

$$ET\left(l, u, \widetilde{\Omega}\right) = \left\{(\mu, t) \mid (\mu, t) \in \widetilde{\Omega} \wedge l \leq t \leq u\right\}$$

with

$$\mathrm{card}_{ET\left(l,u,\widetilde{\Omega}\right)}(\widetilde{\mu}) = \mathrm{card}_{\widetilde{\Omega}}(\widetilde{\mu})$$ □

The effect of an ensure trust operator on query evaluation is illustrated in the following example.

Example 3.2 *Figure 9(a) depicts the operator tree for the graph pattern of the query in Figure 3. The tree is annotated with sample solutions. The ensure trust operator discards the second solution from its input because it only permits*

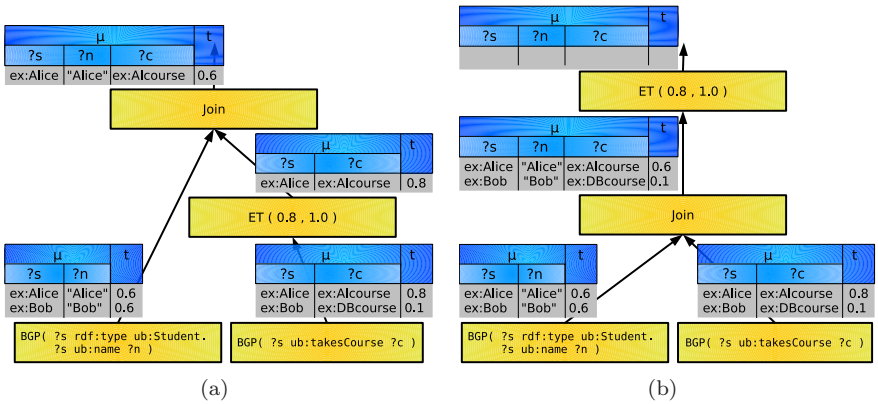

Figure 9: Ensure trust operators in an operator tree with sample solutions

solutions associated with a trust value of at least 0.8. *Hence, in contrast to Figure 5(b), the joined solutions contain only one trust weighted solution mapping.*

Consider a slight variation of the query in Figure 3 where the `ENSURE TRUST` *clause has been defined for the whole group graph pattern (i.e. before the last closing brace in line 8). Figure 9(b) depicts the corresponding operator tree annotated with sample solutions. Notice, the solutions from BGP matching are the same as before (cf. Figure 9(a)). Even so, the overall resulting solutions differ because the ensure trust operator is applied after joining the solutions. Obviously, the position of an* `ENSURE TRUST` *clause in a query pattern matters.* □

3.4 tSPARQL Evaluation Semantics

To define the evaluation semantics of SPARQL queries the SPARQL specification introduces an operation `eval` [PS08, Section 12.5]. The operands of `eval` are the queried dataset and a graph pattern represented by an algebra expression; the result of the operation is a multiset of solution mappings. The definition of `eval` consists of one equation for each algebra symbol; the equations specify the semantics to evaluate an algebra expression with the corresponding algebra operators. For instance, the evaluation of joins is defined as

$$\text{eval}\big(D(G), \text{Join}(P_1, P_2)\big) = \mathit{Join}\Big(\text{eval}\big(D(G), P_1\big), \text{eval}\big(D(G), P_2\big)\Big)$$

where $D(G)$ denotes a dataset D with active graph G and P_1 and P_2 denote the algebra expressions for the joined graph patterns.

The `eval` operation can be extended to define the evaluation semantics of tSPARQL queries. In this case, `eval` returns a multiset of trust weighted solution mappings (instead of a multiset of solution mappings). Under this premise the equations specified in [PS08, Section 12.5] hold with the redefined algebra operators (cf. Section 2.2). However, further equations must be added for the algebra symbols that represent the new clauses (i.e. for `TRUST AS` and

`ENSURE TRUST`):

$$\mathrm{eval}\Big(D(G), \mathrm{PT}(P, v)\Big) = PT\Big(v, \mathrm{eval}\big(D(G), P\big)\Big)$$
$$\mathrm{eval}\Big(D(G), \mathrm{ET}(P, l, u)\Big) = ET\Big(l, u, \mathrm{eval}\big(D(G), P\big)\Big)$$

3.5 Optimization of tSPARQL Query Execution

A well-known heuristic to optimize query execution in relational database systems is selection push-down. Relational algebra expressions are being rewritten to push down selection operators in the operator tree in order to reduce the size of intermediary solutions and, thus, evaluate queries more efficiently. We adapt this heuristic to tSPARQL. In this section we present rewrite rules to push down trust constraints.

Enforcing trust constraints as early as possible may reduce query execution costs by reducing the number of trustweighted solution mappings that have to be processed. However, pre-drawing the evaluation of trust constraints is not as simple as pushing down ensure trust operators: this transformation may modify the semantics of the query unintentionally. In particular, pushing trust constraints in join operations may result in algebra expressions not equivalent to the original expressions. The soundness of rewrite rules that incorporate join operators depends on the trust merge function employed for joins. In the following we focus on rewrite rules that are only valid for the minimum trust merge function $tm_{\min}$.

In the following we say that two algebra expressions P_1 and P_2 are *equivalent*, denoted by $P_1 \equiv P_2$, if $\mathrm{eval}\big(D(G), P_1\big) = \mathrm{eval}\big(D(G), P_2\big)$ for every RDF dataset D with active graph G.

Proposition 3.1 *Let P_1 and P_2 be algebra expressions. For join operators that employ $tm_{\min}$ the following equivalence of algebra expressions holds:*

$$\mathrm{ET}\Big(\mathrm{Join}\big(P_1, P_2\big), l, u\Big) \equiv \mathrm{ET}\Big(\mathrm{Join}\big(\mathrm{ET}(P_1, l, 1), \mathrm{ET}(P_2, l, 1)\big), l, u\Big) \qquad (1)$$

Find the proof of Proposition 3.1 in Appendix B (cf. Proof B.1). The proof idea, however, is the following. For each result $\widetilde{\mu} = (\mu, t)$ of the left term in (1), t is in the interval $[l, u]$; let t_1 and t_2 be the trust value of the two join partners for $\widetilde{\mu}$, respectively; since t is the minimum of t_1 and t_2 both, t_1 and t_2, must be at least l.

Based on Proposition 3.1 we propose to rewrite algebra expressions by replacing terms of the form on the left hand side of (1) by the corresponding term of the form on the right hand side of (1). Furthermore, for left-join operators that employ $tm_{\min}$ we propose a similar rewrite rule based on the following equivalence (for a proof see Proof B.2).

Proposition 3.2 *Let P_1 and P_2 be algebra expressions. For left-join operators that employ $tm_{\min}$ the following equivalence of algebra expressions holds:*

$$\mathrm{ET}\Big(\mathrm{LJoin}\big(P_1, P_2, ex\big), l, u\Big) \equiv \mathrm{ET}\Big(\mathrm{LJoin}\big(\mathrm{ET}(P_1, l, 1), P_2, ex\big), l, u\Big) \qquad (2)$$

To enable an even more extensive push-down of trust constraints we introduce the following equivalences and propose to apply the corresponding rewrite rules.

Proposition 3.3 *Let P be an algebra expressions. The following equivalence of algebra expressions holds:*

$$\mathrm{ET}\left(\mathrm{Filter}(ex, P), l, u\right) \equiv \mathrm{Filter}\left(ex, \mathrm{ET}(P, l, u)\right) \tag{3}$$

Proposition 3.4 *Let P be an algebra expressions. The following equivalence of algebra expressions holds:*

$$\mathrm{ET}\left(\mathrm{PT}(P, v), l, u\right) \equiv \mathrm{PT}\left(\mathrm{ET}(P, l, u), v\right) \tag{4}$$

Proposition 3.5 *Let P be an algebra expressions. The following equivalence of algebra expressions holds:*

$$\mathrm{ET}\left(\mathrm{ET}(P, l_2, u_2), l_1, u_1\right) \equiv \mathrm{ET}\left(P, \max(l_1, l_2), \min(u_1, u_2)\right) \tag{5}$$

Proposition 3.6 *Let P_1 and P_2 be algebra expressions. The following equivalence of algebra expressions holds:*

$$\mathrm{ET}\big(\mathrm{Union}(P_1, P_2), l, u\big) \equiv \mathrm{Union}\big(\mathrm{ET}(P_1, l, u), \mathrm{ET}(P_2, l, u)\big) \tag{6}$$

Proposition 3.7 *Let P_1 and P_2 be algebra expressions. The following equivalence of algebra expressions holds:*

$$\mathrm{ET}\big(\mathrm{Diff}(ex, P_1, P_2), l, u\big) \equiv \mathrm{Diff}\big(ex, \mathrm{ET}(P_1, l, u), P_2\big) \tag{7}$$

Please notice, in contrast to (1) and (2), the equivalences (3) to (7) hold for all trust merge functions; we prove them in Appendix B (cf. Proofs B.3 to B.7).

4 Conclusion

This document specifies the trust-aware query language tSPARQL which is an extended variation of SPARQL. tSPARQL permits to query the trust values in trust weighted RDF graphs as Examples 3.1 and 3.2 illustrate.

References

[ABL+07] Sören Auer, Christian Bizer, Jens Lehmann, Georgi Kobilarov, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a web of open data. In *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007), Busan, South Korea*, volume 4825 of *Lecture Notes in Computer Science*, pages 715–728, Berlin, Heidelberg, November 2007. Springer Verlag.

[BC06] Christian Bizer and Richard Cyganiak. D2R Server - publishing relational databases on the semantic web. Poster at the 5th International Semantic Web Conference (ISWC 2006), November 2006.

[DFJ+04] Li Ding, Timothy W. Finin, Anupam Joshi, Rong Pan, R. Scott Cost, Yun Peng, Pavan Reddivari, Vishal Doshi, and Joel Sachs. Swoogle: A search and metadata engine for the semantic web. In *Proceedings of the 13th ACM Conference on Information and Knowledge Management (CIKM 2004)*, pages 652–659. ACM, November 2004.

[KC04] Graham Klyne and Jeremy J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. W3C Recommendation, February 2004.

[KSGM03] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The Eigentrust algorithm for reputation management in P2P networks. In *Proceedings of the 12th International Conference on World Wide Web (WWW '03)*, pages 640–651, New York, NY, USA, May 2003. ACM Press.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C Recommendation, January 2008.

[TOD07] Giovanni Tummarello, Eyal Oren, and Renaud Delbru. Sindice.com: Weaving the open linked data. In *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007), Busan, South Korea*, volume 4825 of *Lecture Notes in Computer Science*, pages 547–560, Berlin, Heidelberg, November 2007. Springer Verlag.

[XL04] Li Xiong and Ling Liu. PeerTrust: Supporting reputation-based trust in peer-to-peer communities. *IEEE Transactions on Data and Knowledge Engineering, Special Issue on Peer-to-Peer Based Data Management*, 16(7):843–857, July 2004.

[ZL04] Cai-Nicolas Ziegler and Georg Lausen. Spreading activation models for trust propagation. In *Proceedings of the IEEE International Conference on e-Technology, e-Commerce, and e-Service (EEE '04)*, March 2004.

A Changes

This appendix lists the changes between different versions of this document.

December 25, 2008
Revised the English in Section 4.

December 23, 2008
Revised the English in Sections 1 to 3.

December 19, 2008
Added the section about rewrite rules and an appendix with proofs for the propositions employed by the rewrite rules.

March 5, 2008
First complete version of this specification.

B Proofs

Proof B.1 for Proposition 3.1

Let D be an RDF dataset with active graph G; let P_1 and P_2 be algebra expressions; let $\widetilde{\Omega_1} = \text{eval}(D(G), P_1)$ and $\widetilde{\Omega_2} = \text{eval}(D(G), P_2)$.

$$
\begin{aligned}
\text{eval}\Big(D(G), \text{ET}\big(\text{Join}(P_1, P_2), l, u\big)\Big) &= ET\Big(l, u, \text{eval}\big(D(G), \text{Join}(P_1, P_2)\big)\Big) \\
&= ET\Big(l, u, \mathit{Join}(\text{eval}(D(G), P_1), \text{eval}(D(G), P_2))\Big) \\
&= ET\Big(l, u, \mathit{Join}(\widetilde{\Omega_1}, \widetilde{\Omega_2})\Big) \\
&= \Big\{(\mu, t) \mid (\mu, t) \in \mathit{Join}(\widetilde{\Omega_1}, \widetilde{\Omega_2}) \wedge l \leq t \leq u\Big\}
\end{aligned}
$$

$$
\begin{aligned}
= \Big\{(\mu, t) \mid (\mu, t) \in \big\{\big(\mathit{merge}(\mu_1, \mu_2), tm\,(\widetilde{\mu_1}, \widetilde{\mu_2})\big) \mid & \widetilde{\mu_1} = (\mu_1, t_1) \in \widetilde{\Omega_1} \wedge \\
& \widetilde{\mu_2} = (\mu_2, t_2) \in \widetilde{\Omega_2} \wedge \\
& \mu_1 \text{ and } \mu_2 \text{ are compatible}\big\} \wedge\ l \leq t \leq u\Big\}
\end{aligned}
$$

$$
\begin{aligned}
= \Big\{(\mu, t) \mid (\mu, t) \in \big\{\big(\mathit{merge}(\mu_1, \mu_2), tm\,(\widetilde{\mu_1}, \widetilde{\mu_2})\big) \mid & \widetilde{\mu_1} = (\mu_1, t_1) \in \widetilde{\Omega_1} \wedge \\
& \widetilde{\mu_2} = (\mu_2, t_2) \in \widetilde{\Omega_2} \wedge \\
& \mu_1 \text{ and } \mu_2 \text{ are compatible} \wedge \\
& l \leq tm\,(\widetilde{\mu_1}, \widetilde{\mu_2}) \leq u\big\} \wedge\ l \leq t \leq u\Big\}
\end{aligned}
$$

$$
\begin{aligned}
= \Big\{(\mu, t) \mid (\mu, t) \in \big\{\big(\mathit{merge}(\mu_1, \mu_2), tm\,(\widetilde{\mu_1}, \widetilde{\mu_2})\big) \mid & \widetilde{\mu_1} = (\mu_1, t_1) \in \widetilde{\Omega_1} \wedge \\
& \widetilde{\mu_2} = (\mu_2, t_2) \in \widetilde{\Omega_2} \wedge \\
& \mu_1 \text{ and } \mu_2 \text{ are compatible} \wedge \\
& l \leq tm_{\min}\,(\widetilde{\mu_1}, \widetilde{\mu_2}) \leq u\big\} \wedge\ l \leq t \leq u\Big\}
\end{aligned}
$$

$$
\begin{aligned}
&= \Big\{(\mu,t) \mid (\mu,t) \in \big\{\big(merge(\mu_1,\mu_2), tm\,(\widetilde{\mu_1},\widetilde{\mu_2})\big) \mid \widetilde{\mu_1} = (\mu_1,t_1) \in \widetilde{\Omega_1} \wedge \\
&\qquad\qquad \widetilde{\mu_2} = (\mu_2,t_2) \in \widetilde{\Omega_2} \wedge \\
&\qquad\qquad \mu_1 \,\text{and}\, \mu_2 \,\text{are compatible} \wedge \\
&\qquad\qquad l \le t_1 \le 1 \wedge l \le t_2 \le 1\big\} \wedge l \le t \le u\Big\} \\
&= \Big\{(\mu,t) \mid (\mu,t) \in \big\{\big(merge(\mu_1,\mu_2), tm\,(\widetilde{\mu_1},\widetilde{\mu_2})\big) \mid \widetilde{\mu_1} = (\mu_1,t_1) \in \{(\mu_1^*,t_1^*) | (\mu_1^*,t_1^*) \in \widetilde{\Omega_1} \wedge l \le t_1^* \le 1\} \wedge \\
&\qquad\qquad \widetilde{\mu_2} = (\mu_2,t_2) \in \{(\mu_2^*,t_2^*) | (\mu_2^*,t_2^*) \in \widetilde{\Omega_2} \wedge l \le t_2^* \le 1\} \wedge \\
&\qquad\qquad \mu_1 \,\text{and}\, \mu_2 \,\text{are compatible}\big\} \wedge l \le t \le u\Big\} \\
&= \Big\{(\mu,t) \mid (\mu,t) \in \big\{\big(merge(\mu_1,\mu_2), tm\,(\widetilde{\mu_1},\widetilde{\mu_2})\big) \mid \widetilde{\mu_1} = (\mu_1,t_1) \in ET(l,1,\widetilde{\Omega_1}) \wedge \\
&\qquad\qquad \widetilde{\mu_2} = (\mu_2,t_2) \in ET(l,1,\widetilde{\Omega_2}) \wedge \\
&\qquad\qquad \mu_1 \,\text{and}\, \mu_2 \,\text{are compatible}\big\} \wedge l \le t \le u\Big\}
\end{aligned}
$$

$$
\begin{aligned}
&= \Big\{(\mu,t) \mid (\mu,t) \in Join\Big(ET(l,1,\widetilde{\Omega_1}), ET(l,1,\widetilde{\Omega_2})\Big) \wedge l \le t \le u\Big\} \\
&= ET\Big(l,u,Join\big(ET(l,1,\widetilde{\Omega_1}), ET(l,1,\widetilde{\Omega_2})\big)\Big) \\
&= ET\Big(l,u,Join\big(ET(l,1,\mathrm{eval}(D(G),P_1)), ET(l,1,\mathrm{eval}(D(G),P_2))\big)\Big) \\
&= ET\Big(l,u,Join\big(\mathrm{eval}(D(G),\mathrm{ET}(P_1,l,1)), \mathrm{eval}(D(G),\mathrm{ET}(P_2,l,1))\big)\Big) \\
&= ET\Big(l,u,\mathrm{eval}\big(\mathrm{Join}(\mathrm{ET}(P_1,l,1),\mathrm{ET}(P_2,l,1))\big)\Big) \\
&= \mathrm{eval}\Big(D(G),\mathrm{ET}\big(\mathrm{Join}(\mathrm{ET}(P_1,l,1),\mathrm{ET}(P_2,l,1)),l,u\big)\Big)
\end{aligned}
$$

Proof B.2 for Proposition 3.2

Let D be an RDF dataset with active graph G; let P_1 and P_2 be algebra expressions; let $\widetilde{\Omega_1} = \text{eval}(D(G), P_1)$ and $\widetilde{\Omega_2} = \text{eval}(D(G), P_2)$.

$$\begin{aligned}
\text{eval}\Big(D(G), \text{ET}\big(\text{LJoin}(P_1, P_2, ex), l, u\big)\Big) &= ET\Big(l, u, LJoin(\text{eval}(D(G), P_1), \text{eval}(D(G), P_2), ex)\Big)\Big) \\
&= ET\Big(l, u, LJoin(\widetilde{\Omega_1}, \widetilde{\Omega_2}, ex)\Big)\Big) \\
&= ET\Big(l, u, Filter\big(ex, Join(\widetilde{\Omega_1}, \widetilde{\Omega_2})\big) \cup Diff\big(\widetilde{\Omega_1}\big)\Big) \\
&= ET\Big(l, u, Filter\big(ex, Join(\widetilde{\Omega_1}, \widetilde{\Omega_2})\big)\Big) \cup ET\Big(l, u, Diff\big(\widetilde{\Omega_1}\big)\Big)
\end{aligned}$$

Before we continue the proof we introduce the following three lemma.

Lemma 1: $ET\Big(l, u, Join\big(\widetilde{\Omega_1}, \widetilde{\Omega_2}\big)\Big) = ET\Big(l, u, Join\big(ET(l, 1, \widetilde{\Omega_1}), \widetilde{\Omega_2}\big)\Big)$ *– The proof for this lemma is similar to Proof B.1 (cf. page 21).*

Lemma 2: $ET\Big(l, u, Filter\big(ex, Join(\widetilde{\Omega_1}, \widetilde{\Omega_2})\big)\Big) = ET\Big(l, u, Filter\big(ex, Join(ET(l, 1, \widetilde{\Omega_1}), \widetilde{\Omega_2})\big)\Big)$ *– This lemma follows from Lemma 1 and Proof B.3 (cf. page 23).*

Lemma 3: $ET\Big(l, u, Diff\big(\widetilde{\Omega_1}, \widetilde{\Omega_2}, ex\big)\Big) = ET\Big(l, u, Diff\big(ET(l, 1, \widetilde{\Omega_1}), \widetilde{\Omega_2}, ex\big)\Big)$ *– The proof for this lemma is similar to Proof B.7 (cf. page 27).*

With lemma 2 and 3 we continue our proof:

$$\begin{aligned}
&= ET\Big(l, u, Filter\big(ex, Join(ET(l, 1, \widetilde{\Omega_1}), \widetilde{\Omega_2})\big)\Big) \cup ET\Big(l, u, Diff\big(ET(l, 1, \widetilde{\Omega_1}), \widetilde{\Omega_2}, ex\big)\Big) \\
&= ET\Big(l, u, Filter\big(ex, Join(ET(l, 1, \widetilde{\Omega_1}), \widetilde{\Omega_2})\big) \cup Diff\big(ET(l, 1, \widetilde{\Omega_1}), \widetilde{\Omega_2}, ex\big)\Big) \\
&= ET\Big(l, u, LJoin(ET(l, 1, \widetilde{\Omega_1}), \widetilde{\Omega_2}, ex)\Big)\Big) \\
&= ET\Big(l, u, LJoin(ET(l, 1, \text{eval}(D(G), P_1)), \text{eval}(D(G), P_2), ex)\Big)\Big) \\
&= \text{eval}\Big(D(G), \text{ET}\big(\text{LJoin}(\text{ET}(P_1, l, 1), P_2, ex), l, u\big)\Big)
\end{aligned}$$

Proof B.3 for Proposition 3.3

Let D be an RDF dataset with active graph G; let P be an algebra expressions; let $\widetilde{\Omega} = \text{eval}(D(G), P)$.

$$\begin{aligned}
\text{eval}\Big(D(G), \text{ET}\left(\text{Filter}(ex, P), l, u\right)\Big) &= ET\Big(l, u, Filter\big(ex, \text{eval}(D(G), P)\big)\Big) \\
&= ET\Big(l, u, Filter\big(ex, \widetilde{\Omega}\big)\Big) \\
&= \Big\{\, (\mu, t) \;\big|\; (\mu, t) \in Filter\big(ex, \widetilde{\Omega}\big) \wedge l \leq t \leq u \Big\}
\end{aligned}$$

$$
\begin{aligned}
&= \Big\{(\mu,t) \mid (\mu,t) \in \Big\{(\mu^*,t^*) \mid \ (\mu^*,t^*) \in \widetilde{\Omega} \wedge ex(\mu^*) \textit{ is an expression that has an effective boolean value of true}\Big\} \wedge l \leq t \leq u\Big\} \\
&= \Big\{(\mu,t) \mid (\mu,t) \in \Big\{(\mu^*,t^*) \mid \ (\mu^*,t^*) \in \widetilde{\Omega} \wedge ex(\mu^*) \textit{ is an expression that has an effective boolean value of true} \wedge l \leq t^* \leq u\Big\} \wedge l \leq t \leq u\Big\} \\
&= \Big\{(\mu^*,t^*) \mid \ (\mu^*,t^*) \in \widetilde{\Omega} \wedge ex(\mu^*) \textit{ is an expression that has an effective boolean value of true} \wedge l \leq t^* \leq u\Big\} \\
&= \Big\{(\mu^*,t^*) \mid \ (\mu^*,t^*) \in ET(l,u,\widetilde{\Omega}) \wedge ex(\mu^*) \textit{ is an expression that has an effective boolean value of true}\Big\} \\
&= Filter\Big(ex, ET\big(l,u,\widetilde{\Omega}\big)\Big) \\
&= Filter\Big(ex, ET\big(l,u,\mathrm{eval}(D(G),P)\big)\Big) \\
&= \mathrm{eval}\Big(D(G), \mathrm{Filter}\,(ex, \mathrm{ET}(P,l,u))\Big)
\end{aligned}
$$

Proof B.4 for Proposition 3.4
Let D be an RDF dataset with active graph G; let P be an algebra expressions; let $\mathrm{L}(t)$ *be a function that returns an RDF literal of type* `xsd:float` *with the value of t; let* $\widetilde{\Omega} = \mathrm{eval}(D(G),P)$.

$$
\begin{aligned}
\mathrm{eval}\Big(D(G), \mathrm{ET}\,(\mathrm{PT}(P,v),l,u)\Big) &= ET\Big(l,u,PT\big(v,\mathrm{eval}(D(G),P)\big)\Big) \\
&= ET\Big(l,u,PT\big(v,\widetilde{\Omega}\big)\Big) \\
&= \Big\{(\mu,t) \mid (\mu,t) \in PT\big(v,\widetilde{\Omega}\big) \wedge l \leq t \leq u\Big\} \\
&= \Big\{(\mu,t) \mid (\mu,t) \in \Big\{(\mu',t^*) \mid (\mu^*,t^*) \in \widetilde{\Omega} \wedge \mu' = \mu^* \cup \big\{(v,\mathrm{L}(t^*))\big\}\Big\} \wedge l \leq t \leq u\Big\} \\
&= \Big\{(\mu,t) \mid (\mu,t) \in \Big\{(\mu',t^*) \mid (\mu^*,t^*) \in \widetilde{\Omega} \wedge \mu' = \mu^* \cup \big\{(v,\mathrm{L}(t^*))\big\} \wedge l \leq t^* \leq u\Big\} \wedge l \leq t \leq u\Big\} \\
&= \Big\{(\mu',t^*) \mid (\mu^*,t^*) \in \widetilde{\Omega} \wedge \mu' = \mu^* \cup \big\{(v,\mathrm{L}(t^*))\big\} \wedge l \leq t^* \leq u\Big\} \\
&= \Big\{(\mu',t^*) \mid (\mu^*,t^*) \in ET(l,u,\widetilde{\Omega}) \wedge \mu' = \mu^* \cup \big\{(v,\mathrm{L}(t^*))\big\}\Big\} \\
&= PT\Big(v, ET\big(l,u,\widetilde{\Omega}\big)\Big) \\
&= PT\Big(v, ET\big(l,u,\mathrm{eval}(D(G),P)\big)\Big)
\end{aligned}
$$

$$= \text{eval}\Big(D(G), \text{PT}\left(\text{ET}(P, l, u), v\right)\Big)$$

Proof B.5 for Proposition 3.5

Let D be an RDF dataset with active graph G; let P be an algebra expressions; let $\widetilde{\Omega} = \text{eval}(D(G), P)$.

$$\begin{aligned}
\text{eval}\Big(D(G), \text{ET}\left(\text{ET}(P, l_2, u_2), l_1, u_1\right)\Big) &= ET\Big(l_1, u_1, \text{eval}\big(D(G), \text{ET}(P, l_2, u_2)\big)\Big) \\
&= ET\Big(l_1, u_1, ET\big(l_2, u_2, \text{eval}(D(G), P)\big)\Big) \\
&= ET\Big(l_1, u_1, ET\big(l_2, u_2, \widetilde{\Omega}\big)\Big) \\
&= \Big\{(\mu, t) \;\Big|\; (\mu, t) \in ET\big(l_2, u_2, \widetilde{\Omega}\big) \wedge l_1 \leq t \leq u_1\Big\} \\
&= \Big\{(\mu, t) \;\Big|\; (\mu, t) \in \Big\{(\mu^*, t^*) \;\Big|\; (\mu^*, t^*) \in \widetilde{\Omega} \wedge l_2 \leq t^* \leq u_2\Big\} \wedge l_1 \leq t \leq u_1\Big\} \\
&= \Big\{(\mu, t) \;\Big|\; (\mu, t) \in \Big\{(\mu^*, t^*) \;\Big|\; (\mu^*, t^*) \in \widetilde{\Omega} \wedge l_2 \leq t^* \leq u_2 \wedge l_1 \leq t^* \leq u_1\Big\} \wedge l_1 \leq t \leq u_1\Big\} \\
&= \Big\{(\mu^*, t^*) \;\Big|\; (\mu^*, t^*) \in \widetilde{\Omega} \wedge l_2 \leq t^* \leq u_2 \wedge l_1 \leq t^* \leq u_1\Big\} \\
&= \Big\{(\mu^*, t^*) \;\Big|\; (\mu^*, t^*) \in \widetilde{\Omega} \wedge \max(l_1, l_2) \leq t^* \leq \min(u_1, u_2)\Big\} \\
&= ET\big(\max(l_1, l_2), \min(u_1, u_2), \widetilde{\Omega}\big) \\
&= ET\big(\max(l_1, l_2), \min(u_1, u_2), \text{eval}(D(G), P)\big) \\
&= \text{eval}\Big(D(G), \text{ET}\left(P, \max(l_1, l_2), \min(u_1, u_2)\right)\Big)
\end{aligned}$$

Proof B.6 for Proposition 3.6

Let D be an RDF dataset with active graph G; let P_1 and P_2 be algebra expressions; let $\widetilde{\Omega_1} = \text{eval}(D(G), P_1)$ and $\widetilde{\Omega_2} = \text{eval}(D(G), P_2)$.

$$
\begin{aligned}
\text{eval}\Big(D(G), \text{ET}\big(\text{Union}(P_1, P_2), l, u\big)\Big) &= ET\Big(l, u, \text{eval}\big(D(G), \text{Union}(P_1, P_2)\big)\Big) \\
&= ET\Big(l, u, Union(\text{eval}(D(G), P_1), \text{eval}(D(G), P_2))\Big) \\
&= ET\Big(l, u, Union(\widetilde{\Omega_1}, \widetilde{\Omega_2})\Big) \\
&= \Big\{(\mu, t) \mid (\mu, t) \in Union(\widetilde{\Omega_1}, \widetilde{\Omega_2}) \wedge l \leq t \leq u\Big\} \\
&= \Big\{(\mu, t) \mid (\mu, t) \in \widetilde{\Omega_1} \cup \widetilde{\Omega_2} \wedge l \leq t \leq u\Big\} \\
&= \Big\{(\mu, t) \mid (\mu, t) \in \big\{(\mu^*, t^*) | (\mu^*, t^*) \in \widetilde{\Omega_1}\big\} \cup \big\{(\mu^*, t^*) | (\mu^*, t^*) \in \widetilde{\Omega_2}\big\} \wedge l \leq t \leq u\Big\} \\
&= \Big\{(\mu, t) \mid (\mu, t) \in \big\{(\mu^*, t^*) | (\mu^*, t^*) \in \widetilde{\Omega_1} \wedge l \leq t^* \leq u\big\} \cup \big\{(\mu^*, t^*) | (\mu^*, t^*) \in \widetilde{\Omega_2} \wedge l \leq t^* \leq u\big\} \wedge l \leq t \leq u\Big\} \\
&= \Big\{(\mu, t) \mid (\mu, t) \in \big\{(\mu^*, t^*) | (\mu^*, t^*) \in \widetilde{\Omega_1} \wedge l \leq t^* \leq u\big\} \cup \big\{(\mu^*, t^*) | (\mu^*, t^*) \in \widetilde{\Omega_2} \wedge l \leq t^* \leq u\big\}\Big\} \\
&= \Big\{(\mu, t) \mid (\mu, t) \in \big\{(\mu^*, t^*) | (\mu^*, t^*) \in ET(l, u, \widetilde{\Omega_1})\big\} \cup \big\{(\mu^*, t^*) | (\mu^*, t^*) \in ET(l, u, \widetilde{\Omega_2})\big\}\Big\} \\
&= \Big\{(\mu, t) \mid (\mu, t) \in ET(l, u, \widetilde{\Omega_1}) \cup ET(l, u, \widetilde{\Omega_2})\Big\} \\
&= Union\Big(ET\big(l, u, \widetilde{\Omega_1}\big), ET\big(l, u, \widetilde{\Omega_2}\big)\Big) \\
&= Union\Big(ET\big(l, u, \text{eval}(D(G), P_1)\big), ET\big(l, u, \text{eval}(D(G), P_2)\big)\Big) \\
&= \text{eval}\Big(D(G), \text{Union}\big(\text{ET}(P_1, l, u), \text{ET}(P_2, l, u)\big)\Big)
\end{aligned}
$$

Proof B.7 for Proposition 3.7
Let D be an RDF dataset with active graph G; let P_1 and P_2 be algebra expressions; let $\widetilde{\Omega_1} = \text{eval}(D(G), P_1)$ and $\widetilde{\Omega_2} = \text{eval}(D(G), P_2)$.

$$\begin{aligned}
\text{eval}\Big(D(G), \text{ET}\big(\text{Diff}(ex, P_1, P_2), l, u\big)\Big) &= ET\Big(l, u, \text{eval}\big(D(G), \text{Diff}(ex, P_1, P_2)\big)\Big) \\
&= ET\Big(l, u, Diff(\text{eval}(D(G), P_1), \text{eval}(D(G), P_2), ex)\Big) \\
&= ET\Big(l, u, Diff(\widetilde{\Omega_1}, \widetilde{\Omega_2}, ex)\Big) \\
&= \Big\{(\mu, t) \mid (\mu, t) \in Diff(\widetilde{\Omega_1}, \widetilde{\Omega_2}, ex) \wedge l \leq t \leq u\Big\}
\end{aligned}$$

$$\begin{aligned}
= \Big\{(\mu, t) \mid (\mu, t) \in\ & \Big\{(\mu_1, t_1) \mid (\mu_1, t_1) \in \widetilde{\Omega_1} \wedge \textit{for all } (\mu_2, t_2) \in \widetilde{\Omega_2} \textit{ holds: } \mu_1 \textit{ and } \mu_2 \textit{ are not compatible}\Big\} \cup \\
& \Big\{(\mu_1, t_1) \mid (\mu_1, t_1) \in \widetilde{\Omega_1} \wedge \textit{for all } (\mu_2, t_2) \in \widetilde{\Omega_2} \textit{ where } \mu_1 \textit{ and } \mu_2 \textit{ are compatible it holds:} \\
& \qquad ex\Big(merge\big(\mu_1, \mu_2\big)\Big) \textit{ has an effective boolean value of false}\Big\} \\
& \wedge l \leq t \leq u\Big\}
\end{aligned}$$

$$\begin{aligned}
= \Big\{(\mu, t) \mid (\mu, t) \in\ & \Big\{(\mu_1, t_1) \mid (\mu_1, t_1) \in \widetilde{\Omega_1} \wedge \textit{for all } (\mu_2, t_2) \in \widetilde{\Omega_2} \textit{ holds: } \mu_1 \textit{ and } \mu_2 \textit{ are not compatible} \wedge l \leq t_1 \leq u\Big\} \cup \\
& \Big\{(\mu_1, t_1) \mid (\mu_1, t_1) \in \widetilde{\Omega_1} \wedge \textit{for all } (\mu_2, t_2) \in \widetilde{\Omega_2} \textit{ where } \mu_1 \textit{ and } \mu_2 \textit{ are compatible it holds:} \\
& \qquad ex\Big(merge\big(\mu_1, \mu_2\big)\Big) \textit{ has an effective boolean value of false} \wedge l \leq t_1 \leq u\Big\} \\
& \wedge l \leq t \leq u\Big\}
\end{aligned}$$

$$\begin{aligned}
= \Big\{(\mu_1, t_1) \mid\ & (\mu_1, t_1) \in \widetilde{\Omega_1} \wedge \textit{for all } (\mu_2, t_2) \in \widetilde{\Omega_2} \textit{ holds: } \mu_1 \textit{ and } \mu_2 \textit{ are not compatible} \wedge l \leq t_1 \leq u\Big\} \cup \\
& \Big\{(\mu_1, t_1) \mid (\mu_1, t_1) \in \widetilde{\Omega_1} \wedge \textit{for all } (\mu_2, t_2) \in \widetilde{\Omega_2} \textit{ where } \mu_1 \textit{ and } \mu_2 \textit{ are compatible it holds:} \\
& \qquad ex\Big(merge\big(\mu_1, \mu_2\big)\Big) \textit{ has an effective boolean value of false} \wedge l \leq t_1 \leq u\Big\}
\end{aligned}$$

$$= \Big\{(\mu_1, t_1) \mid (\mu_1, t_1) \in ET\big(l, u, \widetilde{\Omega_1}\big) \wedge \textit{for all } (\mu_2, t_2) \in \widetilde{\Omega_2} \textit{ holds: } \mu_1 \textit{ and } \mu_2 \textit{ are not compatible}\Big\} \cup$$
$$\Big\{(\mu_1, t_1) \mid (\mu_1, t_1) \in ET\big(l, u, \widetilde{\Omega_1}\big) \wedge \textit{for all } (\mu_2, t_2) \in \widetilde{\Omega_2} \textit{ where } \mu_1 \textit{ and } \mu_2 \textit{ are compatible it holds: } ex\Big(merge\big(\mu_1, \mu_2\big)\Big) \textit{ has an effective boolean value of false}\Big\}$$

$$= Diff\Big(ET\big(l, u, \widetilde{\Omega_1}\big), \widetilde{\Omega_2}, ex\Big)$$
$$= Diff\Big(ET\big(l, u, \text{eval}(D(G), P_1)\big), \text{eval}(D(G), P_2), ex\Big)$$
$$= \text{eval}\Big(D(G), \big(\text{Diff}(ex, \text{ET}(P_1, l, u), P_2)\big)\Big)$$