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 structured 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 .............................................. 3

2 Trust in SPARQL Query Processing .................... 6
   2.1 Trust-aware Basic Graph Pattern Matching .......... 6
   2.2 Trust-aware Algebra for SPARQL .................. 8

3 SPARQL Extension for Trust Requirements .............. 12
   3.1 tSPARQL Grammar .................................. 12
   3.2 Converting Graph Patterns ......................... 12
   3.3 tSPARQL Algebra .................................. 13
      3.3.1 Project Trust Operator ....................... 13
      3.3.2 Ensure Trust Operator ....................... 15
   3.4 tSPARQL Evaluation Semantics ..................... 16
   3.5 Optimization of tSPARQL Query Execution .......... 17

4 Conclusion .................................................. 18

A Changes ..................................................... 20

B Proofs ......................................................... 21
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 post1. These news only indicate the beginning; content providers such as the review and rating service revyu.com2 or the social tagging service GroupMe3 publish their data in RDF, social network sites such as LiveJournal4 provide FOAF5 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 project6 which creates and publishes RDF representations of various open datasets such as the Wikipedia [ABL+07] and the UniProt7 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

$$tv^C : T \rightarrow \{tv \mid tv \in [-1, 1] \cup \{0\} \}$$

---

1http://blog.sindice.com/?p=5
2http://revyu.com
3http://groupme.org
4http://www.livejournal.com
5http://www.foaf-project.org
6http://esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData
7http://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
0, 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 $\tilde{G}^C$ 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 repre-
sent 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 correspond-
ing 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](image)

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 to-
  gether 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, Q3 to Q6 use trust values in the query. Fundamental to enable information consumers to ask queries such as Q1 to Q6 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.

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

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 Q3 to Q6. Besides accessing trust values a `TRUST AS` clause can even be used to express trust requirements as in queries Q1 and Q2; 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.

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

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 Processing

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 calculated 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 solutions 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 \( \tilde{\mu} \) is a pair \((\mu, t)\) consisting of a solution mapping \(\mu\) (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 \(ta(\tilde{G}^C)\) from the set \(\{t \mid -1 \leq t \leq 1\} \cup \{\emptyset\}\) to a trust weighted RDF graph \(\tilde{G}^C\).
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 \( G^C = (G, tv_C) \) be a trust weighted RDF graph for information consumer \( C \). A trust weighted solution mapping \( (\mu, t) \) is a solution for \( b \) in \( G^C \) if there is a pattern instance mapping \( P \) (as defined in [PS08, Section 12.3.1]) such that \( P(b) \) is a subgraph of \( G \), \( \mu \) is the restriction of \( P \) to the query variables in \( b \), and \( t \) is the aggregated trust value

\[
t = ta\left(P(b)^C\right)
\]

of the trust weighted RDF graph \( P(b)^C = (P(b), tv_C) \) calculated by a trust aggregation function \( ta \). For each solution \( \mu \) for \( b \) \( \text{card}_G(\tilde{\mu}) \) 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

\[
t_{\text{min}}(G, tv_C) = \begin{cases} 
\emptyset & \text{if } \exists t \in G : tv_C(t) = \emptyset \\
\min\{tv_C(t) \mid t \in G\} & \text{else}
\end{cases}
\]

and

\[
t_{\text{avg}}(G, tv_C) = \begin{cases} 
\emptyset & \text{if } \exists t \in G : tv_C(t) = \emptyset \\
\frac{1}{|G|} \sum_{t \in G} tv_C(t) & \text{else}
\end{cases}
\]

**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). \( \mu_1 \) maps \( ?s \) to \text{ex:Alice} and \( ?n \) to the literal \text{Alice}; \( \mu_2 \) maps \( ?s \) to \text{ex:Alice} again and \( ?n \) to the string \text{Bob}. To determine the trust values for both, \( \mu_1 \) and \( \mu_2 \), we choose \( t_{\text{min}} \) 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 \( \mu_1 \) consists of two triples with trust values 0.8 and 0.9, respectively. Hence, our first solution is the trust weighted solution mapping \( \tilde{\mu}_1 = (\mu_1, 0.8) \). For \( \mu_2 \) we have the two trust values 0.8 and 0.1; our second solution is \( \tilde{\mu}_2 = (\mu_2, 0.1) \). □

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
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 \( \tilde{\mu}_1 \) and \( \tilde{\mu}_2 \) is a commutative and associative function that determines a merged trust value \( tm(\tilde{\mu}_1, \tilde{\mu}_2) \) from the set \( \{ t | -1 \leq t \leq 1 \} \cup \{ \emptyset \} \).

Possible trust merge functions are

\[
\begin{align*}
    tm_{\text{min}}(\tilde{\mu}_1, \tilde{\mu}_2) &= \begin{cases} 
        \emptyset & \text{if } t_1 = \emptyset \lor t_2 = \emptyset \\
        \min(t_1, t_2) & \text{else}
    \end{cases} \\
    tm_{\text{avg}}(\tilde{\mu}_1, \tilde{\mu}_2) &= \begin{cases} 
        \emptyset & \text{if } t_1 = \emptyset \lor t_2 = \emptyset \\
        \frac{1}{2}(t_1 + t_2) & \text{else}
    \end{cases}
\end{align*}
\]

where \( \tilde{\mu}_i = (\mu_i, t_i) \).

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

tSPARQL redefines the join operator as follows.

**Definition 2.5** Let \( \tilde{\Omega}_1 \) and \( \tilde{\Omega}_2 \) 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

\[
\text{Join}(\tilde{\Omega}_1, \tilde{\Omega}_2) = \left\{ \text{merge}(\mu_1, \mu_2), \text{tm}(\tilde{\mu}_1, \tilde{\mu}_2) \right\} \left| \tilde{\mu}_1 = (\mu_1, t_1) \in \tilde{\Omega}_1 \land \tilde{\mu}_2 = (\mu_2, t_2) \in \tilde{\Omega}_2 \land \mu_1 \text{ and } \mu_2 \text{ are compatible} \right.
\]

with

\[
\text{card}_{\text{Join}(\tilde{\Omega}_1, \tilde{\Omega}_2)}(\mu) = \sum_{\tilde{\mu}_1 \in \tilde{\Omega}_1, \tilde{\mu}_2 \in \tilde{\Omega}_2} \left\{ \begin{array}{ll}
\text{card}_{\tilde{\Omega}_1}(\tilde{\mu}_1) \cdot \text{card}_{\tilde{\Omega}_2}(\tilde{\mu}_2) & \text{if } \tilde{\mu} = (\mu, t) \text{ with } t = \text{tm}(\tilde{\mu}_1, \tilde{\mu}_2) \text{ and } \\
0 & \text{else}
\end{array} \right.
\]

where \( \text{merge} \) is the merge operation for solution mappings [PS08, Section 12.3] and \( \text{tm} \) 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.

**Example 2.2** The group graph pattern in Figure 5(a) groups two BGP s 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 \( \text{tm}_{\text{min}} \), 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.

---

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

---

9
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 \( \tilde{\Omega} \) 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

\[
Filter\left(ex, \tilde{\Omega}\right) = \left\{ \tilde{\mu} \mid \tilde{\mu} = (\mu, t) \in \tilde{\Omega} \land ex(\mu) \text{ is an expression that has an effective boolean value of true} \right\}
\]

with

\[
\text{card}_{Filter(ex, \tilde{\Omega})}(\tilde{\mu}) = \text{card}_{\tilde{\Omega}}(\tilde{\mu})
\]

**Definition 2.7** Let \( \tilde{\Omega}_1 \) and \( \tilde{\Omega}_2 \) 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

\[
Diff\left(\tilde{\Omega}_1, \tilde{\Omega}_2, ex\right) = \left\{ \tilde{\mu}_1 \mid \tilde{\mu}_1 = (\mu_1, t_1) \in \tilde{\Omega}_1 \land \right.
\]

for all \( \tilde{\mu}_2 = (\mu_2, t_2) \in \tilde{\Omega}_2 \) holds:

\[
\mu_1 \text{ and } \mu_2 \text{ are not compatible}
\]

\[
\left. \cup \left\{ \tilde{\mu}_1 \mid \tilde{\mu}_1 = (\mu_1, t_1) \in \tilde{\Omega}_1 \land \right. \right.
\]

for all \( \tilde{\mu}_2 = (\mu_2, t_2) \in \tilde{\Omega}_2 \) where \( \mu_1 \) and \( \mu_2 \) are compatible it holds:

\[
\text{ex}\left(merge(\mu_1, \mu_2)\right) \text{ has an effective boolean value of false}
\]

with

\[
\text{card}_{Diff(\tilde{\Omega}_1, \tilde{\Omega}_2, ex)}(\tilde{\mu}) = \text{card}_{\tilde{\Omega}_1}(\tilde{\mu})
\]

where \( merge \) is the merge operation for solution mappings [PS08].

**Definition 2.8** Let \( \tilde{\Omega}_1 \) and \( \tilde{\Omega}_2 \) 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

\[
LJoin\left(\tilde{\Omega}_1, \tilde{\Omega}_2, ex\right) = Filter\left(ex, Join(\tilde{\Omega}_1, \tilde{\Omega}_2)\right) \cup Diff\left(\tilde{\Omega}_1, \tilde{\Omega}_2, ex\right)
\]

with

\[
\text{card}_{LJoin(\tilde{\Omega}_1, \tilde{\Omega}_2, ex)}(\tilde{\mu}) = \text{card}_{Filter(ex, Join(\tilde{\Omega}_1, \tilde{\Omega}_2))}(\tilde{\mu}) + \text{card}_{Diff(\tilde{\Omega}_1, \tilde{\Omega}_2, ex)}(\tilde{\mu})
\]

**Definition 2.9** Let \( \tilde{\Omega}_1 \) and \( \tilde{\Omega}_2 \) 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

\[
Union\left(\tilde{\Omega}_1, \tilde{\Omega}_2\right) = \tilde{\Omega}_1 \cup \tilde{\Omega}_2
\]

with

\[
\text{card}_{Union(\tilde{\Omega}_1, \tilde{\Omega}_2)}(\tilde{\mu}) = \text{card}_{\tilde{\Omega}_1}(\tilde{\mu}) + \text{card}_{\tilde{\Omega}_2}(\tilde{\mu})
\]
Definition 2.10 Let $\tilde{\Omega}$ 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

$$\text{ToList}(\tilde{\Omega}) = \left\{ \tilde{\mu} \mid \tilde{\mu} \in \tilde{\Omega} \right\}$$

with

$$\text{card}_{\text{ToList}}(\tilde{\Omega})(\tilde{\mu}) = \text{card}_{\tilde{\Omega}}(\tilde{\mu})$$

Definition 2.11 Let $\tilde{\Psi}$ be a sequence of trust weighted solution mappings; let $\text{cond}$ 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

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

with

$$\text{card}_{\text{OrderBy}}(\tilde{\Psi}, \text{cond})(\tilde{\mu}) = \text{card}_{\tilde{\Psi}}(\tilde{\mu})$$

Definition 2.12 Let $\tilde{\Psi}$ 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

$$\text{Project}(\tilde{\Psi}, PV) = \left\{ \left( \text{proj}(\mu, PV), t \right) \mid \tilde{\mu} = (\mu, t) \in \tilde{\Psi} \right\}$$

with

$$\text{card}_{\text{Project}}(\tilde{\Psi}, PV)(\tilde{\mu}) = \text{card}_{\tilde{\Psi}}(\tilde{\mu})$$

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

Definition 2.13 Let $\tilde{\Psi}$ 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

$$\text{Distinct}(\tilde{\Psi}) = \left\{ \tilde{\mu} \mid \tilde{\mu} \in \tilde{\Psi} \right\}$$

with

$$\text{card}_{\text{Distinct}}(\tilde{\Psi})(\tilde{\mu}) = 1$$

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

Definition 2.14 Let $\tilde{\Psi}$ 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

$$\text{Reduced}(\tilde{\Psi}) = \left\{ \tilde{\mu} \mid \tilde{\mu} \in \tilde{\Psi} \right\}$$

with

$$1 \leq \text{card}_{\text{Reduced}}(\tilde{\Psi})(\tilde{\mu}) \leq \text{card}_{\tilde{\Psi}}(\tilde{\mu})$$

The order of $\text{Reduced}(\tilde{\Psi})$ must preserve any ordering given by $\text{OrderBy}$. □
Definition 2.15 Let $\tilde{\Psi}$ 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

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

The order of $Slice(\tilde{\Psi})$ 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' Vari

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:

\[ PT(Pattern, Variable) \quad \text{and} \quad ET(Pattern, Number, Number) \]

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 \( \tilde{\Omega} \) be a multiset of trust weighted solution mappings; let \( v \) be a query variable which is not bound in any \( \tilde{\mu} \in \tilde{\Omega} \); let \( L(t) \) be a function that returns an RDF literal of type \texttt{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

\[
PT\left(v, \tilde{\Omega}\right) = \left\{ (\mu', t) \mid (\mu, t) \in \tilde{\Omega} \land \mu' = \mu \cup \{ (v, L(t)) \} \right\}
\]
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 \( \text{FILTER(expr)} \)
    \( FS := FS \cup \{ \text{expr} \} \);
  IF \( E \) is of the form \( \text{TRUST AS v} \)
    \( TV := TV \cup \{ v \} \);
  IF \( E \) is of the form \( \text{ENSURE TRUST (l,u)} \)
    \( TC := TC \cup \{ (l,u) \} \);
  IF \( E \) is of the form \( \text{OPTIONAL(P)} \)
    THEN
    Let \( A := \text{Transform(P)} \);
    IF \( A \) is of the form \( \text{Filter(F, A2)} \)
      \( G := \text{LJoin}(G, A2, F) \);
    ELSE
      \( G := \text{LJoin}(G, A, \text{true}) \);
    IF \( E \) is any other form:
    Let \( A := \text{Transform(E)} \);
    \( G := \text{Join}(G, A) \);
  IF \( TC \neq \emptyset \)
    FOR EACH pair \( (l, u) \) in \( TC \)
      \( G := \text{ET}(G, l, u) \);
  IF \( VC \neq \emptyset \)
    FOR EACH variable \( v \) in \( TV \)
      \( G := \text{PT}(G, v) \);
  IF \( FS \neq \emptyset \)
    Let \( X := \text{Conjunction of expressions in } FS \);
    \( G := \text{Filter}(X, G) \);

The result is \( G \).

Figure 7: Adjusted \texttt{Transform} procedure to consider the tSPARQL grammar

with

\[
\text{card}_{\text{PT}(v, \Omega)}(\tilde{\mu}) = \text{card}_{\Omega}(\tilde{\mu})
\]

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

\textbf{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 \( t_m \text{min} \) 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
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 ?т 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 $\bar{\Omega}$ 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

$$ET(l, u, \bar{\Omega}) = \{ (\mu, t) \mid (\mu, t) \in \bar{\Omega} \land l \leq t \leq u \}$$

with

$$\text{card}_{ET(l,u,\bar{\Omega})}(\bar{\mu}) = \text{card}_{\bar{\Omega}}(\bar{\mu})$$

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

$$\text{eval}(D(G), \text{Join}(P_1, P_2)) = \text{Join} \left( \text{eval}(D(G), P_1), \text{eval}(D(G), P_2) \right)$$

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):

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

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 \( t_{\text{min}} \).

In the following we say that two algebra expressions \( P_1 \) and \( P_2 \) are equivalent, denoted by \( P_1 \equiv P_2 \), if \( \text{eval}(D(G), P_1) = \text{eval}(D(G), P_2) \) 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 \( t_{\text{min}} \) the following equivalence of algebra expressions holds:

\[
\text{ET}\left(\text{Join}(P_1, P_2), l, u\right) \equiv \text{ET}\left(\text{Join}(\text{ET}(P_1, l, 1), \text{ET}(P_2, l, 1)), l, u\right)
\]

Find the proof of Proposition 3.1 in Appendix B (cf. Proof B.1). The proof idea, however, is the following. For each result \( \tilde{\mu} = (\mu, t) \) 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 \( \tilde{\mu} \), 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 \( t_{\text{min}} \) 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 \( t_{\text{min}} \) the following equivalence of algebra expressions holds:

\[
\text{ET}\left(\text{LJoin}(P_1, P_2, \text{ex}), l, u\right) \equiv \text{ET}\left(\text{LJoin}(\text{ET}(P_1, l, 1), P_2, \text{ex})), l, u\right)
\]

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:

$$ET(\text{Filter}(ex, P), l, u) \equiv \text{Filter}(ex, ET(P, l, u))$$ (3)

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

$$ET(\text{PT}(P, v), l, u) \equiv \text{PT}(ET(P, l, u), v)$$ (4)

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

$$ET(ET(P, l_2, u_2), l_1, u_1) \equiv ET(P, \max(l_1, l_2), \min(u_1, u_2))$$ (5)

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

$$ET(\text{Union}(P_1, P_2), l, u) \equiv \text{Union}(ET(P_1, l, u), ET(P_2, l, u))$$ (6)

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

$$ET(\text{Diff}(ex, P_1, P_2), l, u) \equiv \text{Diff}(ex, ET(P_1, l, u), P_2)$$ (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


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 $\tilde{\Omega}_1 = \text{eval}(D(G), P_1)$ and $\tilde{\Omega}_2 = \text{eval}(D(G), P_2)$.

\[
\text{eval}(D) \cdot \text{ET}(\text{Join}(P_1, P_2), l, u) = \text{ET}(l, u, \text{eval}(D) \cdot \text{Join}(P_1, P_2))
\]

\[
= \text{ET}(l, u, \text{Join}(\text{eval}(D), \text{eval}(D)))
\]

\[
= \text{ET}(l, u, \text{Join}(\tilde{\Omega}_1, \tilde{\Omega}_2))
\]

\[
= \{(\mu, t) \mid (\mu, t) \in \text{Join}(\tilde{\Omega}_1, \tilde{\Omega}_2) \land l \leq t \leq u\}
\]

\[
= \{(\mu, t) \mid (\mu, t) \in \{(\text{merge}(\mu_1, \mu_2), \text{tm}(\mu_1, \mu_2)) \mid \mu_1 = (\mu_1, t_1) \in \tilde{\Omega}_1 \land \\
\mu_2 = (\mu_2, t_2) \in \tilde{\Omega}_2 \land \\
\mu_1 \text{ and } \mu_2 \text{ are compatible}\} \land l \leq t \leq u\}
\]

\[
= \{(\mu, t) \mid (\mu, t) \in \{(\text{merge}(\mu_1, \mu_2), \text{tm}(\mu_1, \mu_2)) \mid \mu_1 = (\mu_1, t_1) \in \tilde{\Omega}_1 \land \\
\mu_2 = (\mu_2, t_2) \in \tilde{\Omega}_2 \land \\
\mu_1 \text{ and } \mu_2 \text{ are compatible} \land \\
l \leq \text{tm}(\mu_1, \mu_2) \leq u \} \land l \leq t \leq u\}
\]
\[
\{ (\mu, t) \mid (\mu, t) \in \{ (\text{merge}(\mu_1, \mu_2), \text{tm}(\tilde{\mu}_1, \tilde{\mu}_2)) \mid \tilde{\mu}_1 = (\mu_1, t_1) \in \bar{\Omega}_1 \land \\
\tilde{\mu}_2 = (\mu_2, t_2) \in \bar{\Omega}_2 \land \\
\mu_1 \text{ and } \mu_2 \text{ are compatible} \land \\
l \leq t_1 \leq l \leq t_2 \leq 1 \land l \leq t \leq u \} \}
\]

\[
= \{ (\mu, t) \mid (\mu, t) \in \{ (\text{merge}(\mu_1, \mu_2), \text{tm}(\tilde{\mu}_1, \tilde{\mu}_2)) \mid \tilde{\mu}_1 = (\mu_1, t_1) \in \{ (\mu_1^*, \ell_1^*) \mid (\mu_1^*, \ell_1^*) \in \bar{\Omega}_1 \land l \leq \ell_1^* \leq 1 \} \land \\
\tilde{\mu}_2 = (\mu_2, t_2) \in \{ (\mu_2^*, \ell_2^*) \mid (\mu_2^*, \ell_2^*) \in \bar{\Omega}_2 \land l \leq \ell_2^* \leq 1 \} \land \\
\mu_1 \text{ and } \mu_2 \text{ are compatible} \land l \leq t \leq u \} \}
\]

\[
= \{ (\mu, t) \mid (\mu, t) \in \{ (\text{merge}(\mu_1, \mu_2), \text{tm}(\tilde{\mu}_1, \tilde{\mu}_2)) \mid \tilde{\mu}_1 = (\mu_1, t_1) \in \text{ET}(l, 1, \bar{\Omega}_1) \land \\
\tilde{\mu}_2 = (\mu_2, t_2) \in \text{ET}(l, 1, \bar{\Omega}_2) \land \\
\mu_1 \text{ and } \mu_2 \text{ are compatible} \land l \leq t \leq u \} \}
\]

\[
= \{ (\mu, t) \mid (\mu, t) \in \text{Join}(\text{ET}(l, 1, \bar{\Omega}_1), \text{ET}(l, 1, \bar{\Omega}_2)) \land l \leq t \leq u \}
\]

\[
= \text{ET}\left(l, u, \text{Join}(\text{ET}(l, 1, \bar{\Omega}_1), \text{ET}(l, 1, \bar{\Omega}_2))\right)
\]

\[
= \text{ET}\left(l, u, \text{Join}(\text{ET}(l, 1, \text{eval}(D(G), P_1)), \text{ET}(l, 1, \text{eval}(D(G), P_2)))\right)
\]

\[
= \text{ET}\left(l, u, \text{Join}(\text{eval}(D(G), \text{ET}(P_1, l, 1)), \text{eval}(D(G), \text{ET}(P_2, l, 1)))\right)
\]

\[
= \text{ET}\left(l, u, \text{eval}(\text{Join}(\text{ET}(P_1, l, 1), \text{ET}(P_2, l, 1)))\right)
\]

\[
= \text{eval}(D(G), \text{ET}(\text{Join}(\text{ET}(P_1, l, 1), \text{ET}(P_2, l, 1))), l, u)
\]
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 $Ω_1 = \text{eval}(D(G), P_1)$ and $Ω_2 = \text{eval}(D(G), P_2)$.

\[
\text{eval} \left( D(G), \text{ET}(\text{LJoin}(P_1, P_2, \text{ex}), l, u) \right) = \text{ET} \left( l, u, \text{LJoin}(\text{eval}(D(G), P_1), \text{eval}(D(G), P_2), \text{ex}) \right) \\
= \text{ET} \left( l, u, \text{LJoin}(Ω_1, Ω_2, \text{ex}) \right) \\
= \text{ET} \left( l, u, \text{Filter}(\text{ex}, \text{Join}(Ω_1, Ω_2)) \cup \text{Diff}(Ω_1) \right) \\
= \text{ET} \left( l, u, \text{Filter}(\text{ex}, \text{Join}(Ω_1, Ω_2)) \cup \text{ET}(l, u, \text{Diff}(Ω_1)) \right)
\]

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

**Lemma 1:** $\text{ET}(l, u, \text{Join}(Ω_1, Ω_2)) = \text{ET}(l, u, \text{Join}(\text{ET}(l, 1, Ω_1), Ω_2))$ – The proof for this lemma is similar to Proof B.1 (cf. page 21).

**Lemma 2:** $\text{ET}(l, u, \text{Filter}(\text{ex}, \text{Join}(Ω_1, Ω_2))) = \text{ET}(l, u, \text{Filter}(\text{ex}, \text{Join}(\text{ET}(l, 1, Ω_1), Ω_2)))$ – This lemma follows from Lemma 1 and Proof B.3 (cf. page 23).

**Lemma 3:** $\text{ET}(l, u, \text{Diff}(Ω_1, Ω_2, \text{ex})) = \text{ET}(l, u, \text{Diff}(\text{ET}(l, 1, Ω_1), Ω_2, \text{ex}))$ – The proof for this lemma is similar to Proof B.7 (cf. page 27).

With lemma 2 and 3 we continue our proof:

\[
= \text{ET} \left( l, u, \text{Filter}(\text{ex}, \text{Join}(\text{ET}(l, 1, Ω_1), Ω_2)) \right) \cup \text{ET} \left( l, u, \text{Diff}(\text{ET}(l, 1, Ω_1), Ω_2, \text{ex}) \right) \\
= \text{ET} \left( l, u, \text{Filter}(\text{ex}, \text{Join}(\text{ET}(l, 1, Ω_1), Ω_2)) \cup \text{Diff}(\text{ET}(l, 1, Ω_1), Ω_2, \text{ex}) \right) \\
= \text{ET} \left( l, u, \text{LJoin}(\text{ET}(l, 1, Ω_1), Ω_2, \text{ex}) \right) \\
= \text{ET} \left( l, u, \text{LJoin}(\text{ET}(l, 1, \text{eval}(D(G), P_1)), \text{eval}(D(G), P_2), \text{ex})) \right) \\
= \text{eval} \left( D(G), \text{ET}(\text{LJoin}(P_1, P_2, \text{ex}), l, u) \right)
\]

Proof B.3 for Proposition 3.3

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

\[
\text{eval} \left( D(G), \text{ET}(\text{Filter}(\text{ex}, P), l, u) \right) = \text{ET} \left( l, u, \text{Filter}(\text{ex}, \text{eval}(D(G), P)) \right) \\
= \text{ET} \left( l, u, \text{Filter}(\text{ex}, Ω) \right) \\
= \left\{ (μ, t) \mid (μ, t) \in \text{Filter}(\text{ex}, Ω) \land l \leq t \leq u \right\}
\]
$\{ (\mu, t) \mid (\mu, t) \in \Omega \land ex(\mu^*) \text{ is an expression that has an effective boolean value of true} \}$

$\{ (\mu, t) \mid (\mu, t) \in \Omega \land ex(\mu^*) \text{ is an expression that has an effective boolean value of true} \land 1 \leq t \leq u \}$

$\{ (\mu^*, t^*) \mid (\mu^*, t^*) \in \Omega \land ex(\mu^*) \text{ is an expression that has an effective boolean value of true} \land 1 \leq t^* \leq u \}$

$\{ (\mu^*, t^*) \mid (\mu^*, t^*) \in ET(l, u, \Omega) \land ex(\mu^*) \text{ is an expression that has an effective boolean value of true} \}$

$\text{Filter} \left( \text{ex}, ET(l, u, \Omega) \right)$

$\text{Filter} \left( \text{ex}, ET(l, u, \text{eval}(D(G), P)) \right)$

$\text{eval} \left( D(G), \text{Filter} \left( \text{ex}, ET(P, l, u) \right) \right)$

**Proof B.4 for Proposition 3.4**

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

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

$= ET \left( l, u, PT(v, \Omega) \right)$

$= \{ (\mu, t) \mid (\mu, t) \in \Omega \land 1 \leq t \leq u \}$

$= \{ (\mu, t) \mid (\mu, t) \in \{ (\mu', t^*) \mid (\mu', t^*) \in \Omega \land \mu' = \mu^* \cup \{ v, L(t^*) \} \} \land 1 \leq t \leq u \}$

$= \{ (\mu, t) \mid (\mu, t) \in \{ (\mu', t^*) \mid (\mu', t^*) \in \Omega \land \mu' = \mu^* \cup \{ v, L(t^*) \} \land 1 \leq t^* \leq u \}$

$= PT \left( v, ET \left( l, u, \Omega \right) \right)$

$= PT \left( v, ET \left( l, u, \text{eval}(D(G), P) \right) \right)$
Proof B.5 for Proposition 3.5

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

$$\text{eval}(D(G), \text{ET}(P, l_2, u_2)) = \text{ET}(l_1, u_1, \text{eval}(D(G), \text{ET}(P, l_2, u_2)))$$

$$= \text{ET}(l_1, u_1, \text{eval}(D(G), P))$$

$$= \text{ET}(l_1, u_1, \text{ET}(l_2, u_2, \tilde{\Omega}))$$

$$= \{ (\mu, 0) \mid (\mu, t) \in \text{ET}(l_2, u_2, \tilde{\Omega}) \land l_1 \leq t \leq u_1 \}$$

$$= \{ (\mu, 0) \mid (\mu, t) \in \{ (\mu^*, t^*) \mid (\mu^*, t^*) \in \tilde{\Omega} \land l_2 \leq t^* \leq u_2 \} \land l_1 \leq t \leq u_1 \}$$

$$= \{ (\mu^*, t^*) \mid (\mu^*, t^*) \in \tilde{\Omega} \land l_2 \leq t^* \leq u_2 \land l_1 \leq t^* \leq u_1 \}$$

$$= \{ (\mu^*, t^*) \mid (\mu^*, t^*) \in \tilde{\Omega} \land \max(l_1, l_2) \leq t^* \leq \min(u_1, u_2) \}$$

$$= \text{ET}(\max(l_1, l_2), \min(u_1, u_2), \tilde{\Omega})$$

$$= \text{ET}(\max(l_1, l_2), \min(u_1, u_2), \text{eval}(D(G), P))$$

$$= \text{eval}(D(G), \text{ET}(P, \max(l_1, l_2), \min(u_1, u_2)))$$
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 $\tilde{\Omega}_1 = \text{eval}(D(G), P_1)$ and $\tilde{\Omega}_2 = \text{eval}(D(G), P_2)$.

$$
\text{eval} \left( D(G), \text{ET} \left( \text{Union}(P_1, P_2), l, u \right) \right) = \text{ET} \left( l, u, \text{eval}(D(G), \text{Union}(P_1, P_2)) \right)
$$

$$
= \text{ET} \left( l, u, \text{Union}(\tilde{\Omega}_1, \tilde{\Omega}_2) \right)
$$

$$
\{ \mu, t \mid \mu, t \in \tilde{\Omega}_1 \cup \tilde{\Omega}_2 \land l \leq t \leq u \}
$$

$$
\{ \mu, t \mid \mu, t \in \{ (\mu^*, t^*) | (\mu^*, t^*) \in \tilde{\Omega}_1 \} \cup \{ (\mu^*, t^*) | (\mu^*, t^*) \in \tilde{\Omega}_2 \} \land l \leq t \leq u \}
$$

$$
\{ \mu, t \mid \mu, t \in \{ (\mu^*, t^*) | (\mu^*, t^*) \in \tilde{\Omega}_1 \land l \leq t \leq u \} \cup \{ (\mu^*, t^*) | (\mu^*, t^*) \in \tilde{\Omega}_2 \land l \leq t \leq u \} \land l \leq t \leq u \}
$$

$$
\{ \mu, t \mid \mu, t \in \{ (\mu^*, t^*) | (\mu^*, t^*) \in \text{ET}(l, u, \tilde{\Omega}_1) \} \cup \{ (\mu^*, t^*) | (\mu^*, t^*) \in \text{ET}(l, u, \tilde{\Omega}_2) \} \}
$$

$$
\{ \mu, t \mid (\mu, t) \in \text{ET}(l, u, \tilde{\Omega}_1) \cup \text{ET}(l, u, \tilde{\Omega}_2) \}
$$

$$
= \text{Union} \left( \text{ET}(l, u, \tilde{\Omega}_1), \text{ET}(l, u, \tilde{\Omega}_2) \right)
$$

$$
= \text{Union} \left( \text{ET}(l, u, \text{eval}(D(G), P_1)), \text{ET}(l, u, \text{eval}(D(G), P_2)) \right)
$$

$$
= \text{eval} \left( D(G), \text{Union}(P_1, l, u), \text{ET}(P_2, l, u) \right)
$$
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 $\Omega_1 = \text{eval}(D(G), P_1)$ and $\Omega_2 = \text{eval}(D(G), P_2)$.

\[
\text{eval}\left(D(G), \text{ET}(\text{Diff}(\text{ex}, P_1, P_2), l, u)\right) = \text{ET}\left(l, u, \text{eval}(D(G), \text{Diff}(\text{ex}, P_1, P_2))\right)
\]
\[
= \text{ET}\left(l, u, \text{eval}(D(G), \text{Diff}(\Omega_1, \Omega_2, \text{ex}))\right)
\]
\[
= \left\{(\mu, t) \mid (\mu, t) \in \text{Diff}(\Omega_1, \Omega_2, \text{ex}) \wedge l \leq t \leq u\right\}
\]

\[
= \left\{(\mu, t) \mid (\mu, t) \in \Omega_1 \wedge \text{for all } (\mu_2, t_2) \in \Omega_2 \text{ holds: } \mu_1 \text{ and } \mu_2 \text{ are not compatible}\right\} \cup
\]
\[
\left\{(\mu, t) \mid (\mu_1, t_1) \in \Omega_1 \wedge \text{for all } (\mu_2, t_2) \in \Omega_2 \text{ where } \mu_1 \text{ and } \mu_2 \text{ are compatible it holds: ex}(\text{merge}(\mu_1, \mu_2)) \text{ has an effective boolean value of false}\right\}
\]

\[
= \left\{(\mu, t) \mid (\mu, t) \in \Omega_1 \wedge \text{for all } (\mu_2, t_2) \in \Omega_2 \text{ holds: } \mu_1 \text{ and } \mu_2 \text{ are not compatible } \wedge l \leq t \leq u\right\} \cup
\]
\[
\left\{(\mu, t) \mid (\mu_1, t_1) \in \Omega_1 \wedge \text{for all } (\mu_2, t_2) \in \Omega_2 \text{ where } \mu_1 \text{ and } \mu_2 \text{ are compatible it holds: ex}(\text{merge}(\mu_1, \mu_2)) \text{ has an effective boolean value of false } \wedge l \leq t \leq u\right\}
\]
= \{(\mu_1, t_1) \mid (\mu_1, t_1) \in ET(l, u, \Omega_1) \land \text{for all} (\mu_2, t_2) \in \Omega_2 \text{ holds: } \mu_1 \text{ and } \mu_2 \text{ are not compatible}\} \cup \\
= \{(\mu_1, t_1) \mid (\mu_1, t_1) \in ET(l, u, \Omega_1) \land \text{for all} (\mu_2, t_2) \in \Omega_2 \text{ where } \mu_1 \text{ and } \mu_2 \text{ are compatible it holds: } \\
\quad \quad \quad \quad \quad ex(merge(\mu_1, \mu_2)) \text{ has an effective boolean value of false}\} \\
= \text{Diff}(ET(l, u, \Omega_1), \Omega_2, ex) \\
= \text{Diff}(ET(l, u, \text{eval}(D(G), P_1)), \text{eval}(D(G), P_2), ex) \\
= \text{eval}(D(G), (\text{Diff}(ex, ET(P_1, l, u), P_2)))