Querying Trust in RDF Data with tSPARQL

Olaf Hartig

Humboldt-Universität zu Berlin
`hartig@informatik.hu-berlin.de`

Abstract. Today a large amount of RDF data is published on the Web. However, the openness of the Web and the ease to combine RDF data from different sources creates new challenges. The Web of data is missing a uniform way to assess and to query the trustworthiness of information. In this paper we present tSPARQL, a trust-aware extension to SPARQL. Two additional keywords enable users to describe trust requirements and to query the trustworthiness of RDF data. Hence, tSPARQL allows adding trust to RDF-based applications in an easy manner. As the foundation we propose a trust model that associates RDF statements with trust values and we extend the SPARQL semantics to access these trust values in tSPARQL. Furthermore, we discuss opportunities to optimize the execution of tSPARQL queries.

1 Introduction

During recent years a large amount of data described by RDF has been published on the Web; large datasets are interlinked; new applications emerge which utilize this data in novel and innovative ways. However, the openness of the Web and the ease to combine RDF data from different sources creates new challenges for applications. Unreliable data could dominate results of queries, taint inferred data, affect knowledge bases, and have negative or misleading impact on software agents. Hence, questions of reliability and trustworthiness must be addressed. While several approaches consider trustworthiness of potential sources of data (e.g. [1,2,3]), little has been done considering the actual data itself.

What is missing for is a uniform way to rate the trustworthiness of the data on the Web and standardized mechanisms to access and to use these ratings. Users as well as software agents have to be able to utilize trust ratings and base their decisions upon them. They have to be enabled to ask queries such as:

Q_1 Return a list of garages close to a specific location ordered by the trustworthiness of the data.
Q_2 Return trustworthy reviews for a specific restaurant.
Q_3 Return the most trustworthy review for each hotel in the city of Heraklion.

To ask queries like these this paper presents appropriate extensions for RDF and its query language SPARQL. Our main contributions are:

– a trust model for RDF data which associates triples with trust values, and
– a trust-aware query language, tSPARQL, which extends SPARQL to describe trust requirements and to access the trustworthiness of query solutions.

This paper is structered as follows. First, Section 2 outlines our trust model for RDF. In Section 3 we informally present tSPARQL by introducting two new query clauses that enable users to ask queries such as Q_1 to Q_3. In the remainder we discuss the necessary extensions for tSPARQL in detail. We present our approach for trust-aware processing of queries (Section 4); followed by the extensions for the new clauses (Section 5) and a discussion of opportunities to optimize the execution of tSPARQL queries (Section 6). An evaluation of our approach is given in Section 7. Finally, we review related work in Section 8 and conclude in Section 9.

2 A Trust Model for RDF

As the foundation of a trust infrastructure that considers the trustworthiness of RDF data we introduce a trust model for RDF. The goal of our trust model is to rate information expressed in RDF according to trustworthiness. Since information expressed in RDF is represented by triples our model represents the trustworthiness of triples. Our fundamental understanding of the *trustworthiness of RDF triples* is the subjective belief or disbelief in the truth of the information represented by the triples. Notice, disbelief is a negative form of belief: disbelief in the truth of information is belief in the untruth of this information; i.e., disbelief is the belief that the information is false. Since belief is a personal attitude the trustworthiness of triples depends on the information consumer.

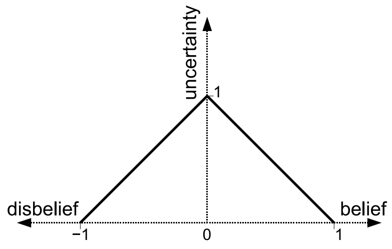


Fig. 1. Meaning of trust values

To enable machine-based processing we introduce a quantifiable measure; we represent the trustworthiness of RDF triples by a *trust value* which is either unknown or a value in the interval [-1,1]. We define the meaning of these values by a specification of the interval boundaries: a trust value of 1 represents absolute belief in the information represented by the corresponding triples; -1 represents absolute disbelief; intermediary values represent degrees of belief/disbelief. We understand the difference between trust values and the extrema 1 and -1 as uncertainty (cf. Figure 1). For a value of 1 the consumer is absolutely sure about the truth of the corresponding triples; a positive value less than 1 still represents belief in the truth; however, to a certain degree the consumer is unsure regarding the assessment. Hence, the lower the trust value, the greater the uncertainty. A value of 0, finally, represents absolute uncertainty. The same holds for negative trust values with respect to disbelief: the higher the negative trust value, the greater the uncertainty. Hence, absolute uncertainty regarding a truth assessment, i.e. the value 0, is equal to the lack of belief as well as the lack of disbelief. Furthermore, we permit unknown trust values, denoted by $\emptyset$, for cases where it is impossible to determine the trustworthiness of triples. Please note the significant difference between a trust value of 0 and an unknown trust value; while the latter denotes the trust management system has no information, a value of 0 represents the expressed lack of belief/disbelief. To determine trust values we define a *trust function.*

Definition 1. *Let T be the set of all RDF triples. A* ***trust function*** tv^C *for RDF triples is a mapping* $\mathrm{tv}^C : T \rightarrow \{tv \mid tv \in [\text{-}1, 1]\} \cup \{\emptyset\}$ *which assigns every triple in T a subjective trust value that represents the trustworthiness of the triple specific to an information consumer C.*

Our trust model does not prescribe a specific implementation of determining trust values. Instead, we allow each system to provide its own, application-specific trust function. Determining trust values may be based on provenance information [4] as in the Inference Web trust component [5]; the TRELLIS system [6] additionally considers related data; the FilmTrust application [7] combines provenance information and recommendations from other consumers.

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

Definition 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. Figure 2 depicts a trust weighted RDF graph; The edges represent the predicates of triples. They are annotated with the predicate identifier as usual and with an additional label for the consumer-specific trust value of the corresponding triple. One of the triples in the graph asserts that resource `ex1:Kastro` is a hotel building; this triple is associated with a trust value of 0.95. □

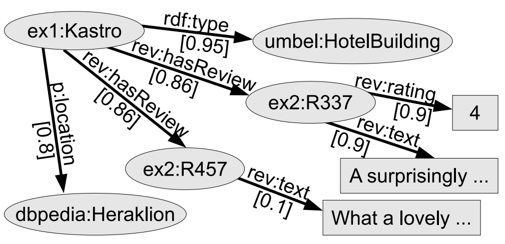


Fig. 2. A trust weighted RDF graph

In addition to the trustworthiness of single triples we represent the trustworthiness of sets of triples, i.e. of whole RDF graphs. The trustworthiness of an RDF graph is an aggregation of the trustworthiness of its triples. Hence, we introduce a *trust aggregation function* to calculate the trust value for RDF graphs.

Definition 3. *A* ***trust aggregation function*** *for trust weighted RDF graphs is a function* ta *which assigns each trust weighted RDF graph $\widetilde{G}^C$ an aggregated trust value* $\mathrm{ta}\big(\widetilde{G}^C\big)$ *that represents the trustworthiness of $\widetilde{G}^C$.*

As with the trust function we do not prescribe a specific aggregation function. Applications have the freedom to choose a function that fits their use cases. The minimum, for instance, is a cautious choice; it assumes the trustworthiness of a set of triples is only as trustworthy as the least trusted triple. The median, a more optimistic choice, is another reasonable trust aggregation function. However, each trust aggregation function must process unknown trust values in a meaningful way. A possible approach is to return an unknown value if an input value is unknown. We will investigate aggregation of trust values more closely in the future.

Please notice, we see the primary usage of trust values for triples within the trust component of a system; trust values are parts of data structures that

```
SELECT ?garage ?t WHERE {
  ?garage a <http://umbel.org/umbel/sc/AutoRepairShop>;
          geo:lat ?lat; geo:long ?long .
  FILTER ( ex:distance(?lat,?long,35.33,25.13) < 10.0 )
  TRUST AS ?t
}
ORDER BY ?t
```

Fig. 3. tSPARQL representation of query Q_1 (prefix declarations omitted)

represent trust weighted RDF graphs. However, to explicitly assert the trustworthiness of RDF data we provide a trust vocabulary[1].

3 Accessing Trust Values in Queries

Trust ratings for RDF data add little value if they cannot be utilized while accessing the data. Therefore, we propose tSPARQL as an extension to the RDF query language SPARQL [8]. SPARQL is of declarative nature; evaluation is based on graph patterns that must match subgraphs in the queried RDF graphs. With tSPARQL users can additionally access trust values that represent the trustworthiness of matching subgraphs. This additional expressivity allows for queries such as Q_1 to Q_3. To express these queries we add a `TRUST AS` clause to the query language. Consider the query in Figure 3 which expresses Q_1 in tSPARQL. The query contains a `TRUST AS` clause with a new variable `?t` which allows access to the trust values of the subgraphs that match the query pattern.

The `TRUST AS` clause offers the following novel features: i) the new variable can become part of the query result, ii) it can be used for sorting the results, iii) it can be associated with parts of the query pattern, and iv) two variables that represent trust values of different query pattern parts can be compared. This approach can even be used for expressing requirements regarding the trustworthiness of query results as in query Q_2: in addition to the `TRUST AS` clause users simply add a `FILTER` clause which restricts the new variable. However, for convenience we propose another new clause for these cases, namely the `ENSURE TRUST` clause which includes a pair of numbers that denote a lower bound and an upper bound, respectively. Figure 4 depicts the tSPARQL representation of Q_2; due to the `ENSURE TRUST` clause only those solutions become part of the result where the trust value of the matching subgraph is at least 0.5 and at most 1.0.

Queries Q_1 and Q_2 access the trust values of the subgraphs that match the whole query pattern. Query Q_3, in contrast, compares the trustworthiness of different parts of each query result. Hence, for Q_3 we must limit the scope of `TRUST AS` clauses to subgraphs that match only parts of the whole query pattern. This limitation can easily be expressed by associating the clause with a specific part of the query pattern as the tSPARQL representation of Q_3 in Figure 5 illustrates. The `TRUST AS` clause in line 6, for instance, solely binds variable ?t1 with trust values of subgraphs that match the graph pattern in line 5. Hence, the position of `TRUST AS` clauses in the query pattern matters; the same holds for `ENSURE TRUST`.

[1] http://trdf.sourceforge.net/trustvocab

```
SELECT ?r ?text WHERE {
  <http://dbpedia.org/resource/The_Cherry_Street_Tavern>
      rev:hasReview ?r .
  ?r rev:text ?text .
  ENSURE TRUST ( 0.5, 1.0 )
}
```

Fig. 4. tSPARQL representation of query Q_2 (prefix declarations omitted)

To enable the proposed extensions we must enhance SPARQL in two ways. First, we extend the query language with the new clauses and define operations for them. Second, since trust values are currently not part of SPARQL we also extend the processing of SPARQL queries to consider trust values. In the remainder of this paper we describe the details of our extensions.

```
SELECT ?h ?txt1 WHERE {
  ?h rdf:type <http://umbel.org/umbel/sc/HotelBuilding> ;
     p:location <http://dbpedia.org/resource/Heraklion> .
  { ?hotel rev:hasReview [ rev:text ?txt1 ]
     TRUST AS ?t1 }
  OPTIONAL { ?h rev:hasReview [ rev:text ?txt2 ]
                 TRUST AS ?t2 FILTER ( ?t2 > ?t1 ) }
  FILTER ( ! BOUND (?txt2) )
}
```

Fig. 5. tSPARQL representation of query Q_3 (prefix declarations omitted)

4 Trust in SPARQL Query Processing

The semantics of SPARQL do not consider trust values. To implement trust-aware query processing we redefine parts of the semantics for tSPARQL. In this section, we first take a brief look at SPARQL query processing and query evaluation; afterwards we present our adaptations.

The SPARQL specification [8] 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. Finally, a result form definition specifies how to create the query result from the solutions. This operational definition of the semantics forms the basis for query processing.

To consider trustworthiness during query processing and to enable operators to access trust values we extend query evaluation for tSPARQL. First, we redefine the notion of solutions because the trust values have to become part of the solutions. Second, we specify how the algebra operates on the extended solutions.

4.1 Trust-aware Basic Graph Pattern Matching

SPARQL is based on graph patterns and subgraph matching. The elementary graph pattern is called a *basic graph pattern* (BGP); it is a set of *triple patterns* which are RDF triples that may contain variables at the subject, predicate, and object position. During evaluation values are bound to the variables according to the subgraphs that match BGPs. These variable bindings are called *solution mapping*. Besides variables SPARQL permits blank nodes in triple patterns. Blank nodes have to be associated with values during pattern matching similiar to variables. Hence, the SPARQL specification additionally introduces *RDF instance mapping*s that map blank nodes to values. A *solution* for a BGP is each solution mapping which, in combination with an RDF instance mapping, maps the BGP to a subgraph of the queried RDF graph. The result of BGP matching is a multiset (or bag) of solution mappings; a solution mapping can be combined with different RDF instance mappings to map the BGP to different subgraphs.

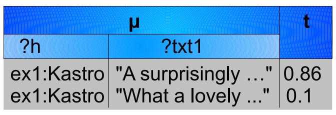
We adapt the definitions for tSPARQL. However, we associate each solution mapping with a trust value.

Definition 4. *A* ***trust weighted solution mapping*** $\widetilde{\mu}$ *is a pair* (μ, t) *consisting of a solution mapping* μ *and a trust value* t*. We denote the cardinality of* $\widetilde{\mu}$ *in a multiset* $\widetilde{\Omega}$ *of trust weighted solution mappings with* $\mathrm{card}_{\widetilde{\Omega}}(\widetilde{\mu})$.

Following the definition from the SPARQL specification, we define solutions for a BGP over a trust weighted RDF graph. Every solution mapping that is a solution to a BGP represents one matching subgraph; the trust value of this solution mapping must represent the trustworthiness of the subgraph; hence, the trust value can be calculated by a trust aggregation function (cf. Definition 3):

Definition 5. *Let* b *be a BGP; let* $\widetilde{G}^C = (G, tv^C)$ *be a trust weighted RDF graph. The trust weighted solution mapping* (μ, t) *is a* ***solution*** *for* b *in* $\widetilde{G}^C$ *if there is an RDF instance mapping* σ *such that i)* $\mu(\sigma(b))$ *is a subgraph of* G*, ii)* μ *is a mapping for the query variables in* b*, and iii)* $t = \mathrm{ta}(\widetilde{S}^C)$ *is the aggregated trust value of the trust weighted RDF graph* $\widetilde{S}^C = \left(\mu(\sigma(b)), tv^C\right)$ *calculated by trust aggregation function* ta*. For each solution* μ *for* b $\mathrm{card}_{\widetilde{\Omega}}(\widetilde{\mu})$ *is the number of distinct RDF instance mappings* σ *such that* $\mu(\sigma(b))$ *is a subgraph of* G*.*

With our definition of solution the result of BGP matching is a multiset of trust weighted solution mappings. Since the solutions are calculated for a trust weighted RDF graph they are calculated in the context of a specific information consumer. BGP matching in the context of another consumer may yield solutions with different trust values because assessing the trustworthiness of matching subgraphs is subjective; hence, the trust values associated with the triples of matching subgraphs are consumer-specific.



μ		t
?h	?txt1	
ex1:Kastro	"A surprisingly ..."	0.86
ex1:Kastro	"What a lovely ..."	0.1

Fig. 6. Trust weighted solution

Example 2. When we apply the BGP in line 5 of Figure 5 to our sample trust weighted RDF graph in Figure 2 we find two matching subgraphs resulting in the two solutions shown in Figure 6. μ_1 maps `?h` to `ex1:Kastro` and `?txt1` to the literal "A surprisingly ..."; μ_2 maps `?h` to `ex1:Kastro` and `?txt1` to "What a lovely ... ." To determine the trust values for both, μ_1 and μ_2, we choose the minimum as our application-specific trust aggregation function. The subgraph for μ_1 consists of two triples with trust values 0.86 and 0.9. Hence, our first solution is the trust weighted solution mapping $\widetilde{\mu_1} = (\mu_1, 0.86)$. For μ_2 we have the two trust values 0.86 and 0.1; our second solution is $\widetilde{\mu_2} = (\mu_2, 0.1)$. $\square$

4.2 Enhanced SPARQL Algebra

After defining trust weighted solution mappings we now explain 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 our new clauses (cf. Section 3) we need new types of operators. To enable these new operators to access the trust values in solutions all operators have to consider the trust values. Hence, for tSPARQL we redefine the conventional SPARQL algebra operators to operate on multisets of trust weighted solution mappings. In the following we exemplarily present the redefined join operator.

The conventional join operator represents a group graph pattern. The two operands of the operator are multisets of solution mappings. Every mapping from one operand is merged with every mapping from the other if they are compatible. Solution mappings are *compatible* if all variables specified in both mappings are bound to the same values. *Merging* two solution mappings combines all variable bindings from both mappings in a new one.

A join operator that operates on trust weighted solution mappings has to consider the trust values while merging solutions. The trust value of a merged solution mapping is an aggregation of the trust values associated with the individual mappings that has been used for merging. For this purpose we introduce another aggregation function which we call *trust merge function.*

Definition 6. *A **trust merge function** for two trust weighted solution mappings $\widetilde{\mu_1}$ and $\widetilde{\mu_2}$ is a commutative and associative function* tm *that determines a merged trust value* $\mathrm{tm}(\widetilde{\mu_1}, \widetilde{\mu_2})$.

We notice that trust merge functions must be commutative and associative because the join operator is a commutative and associative operation. Using trust merge functions we redefine the join operator.

Definition 7. *Let $\widetilde{\Omega_1}$ and $\widetilde{\Omega_2}$ be multisets of trust weighted solution mappings; let* merge *be the merge operation for solution mappings [8]. The result of a **join operator** is a multiset of trust weighted solution mappings which is defined as*

$$\begin{aligned} Join(\widetilde{\Omega_1}, \widetilde{\Omega_2}) = \big\{ \big(\mathrm{merge}(\mu_1, \mu_2), \mathrm{tm}(\widetilde{\mu_1}, \widetilde{\mu_2})\big) \,\big|\, & \widetilde{\mu_1} = (\mu_1, t_1) \in \widetilde{\Omega_1} \wedge \\ & \widetilde{\mu_2} = (\mu_2, t_2) \in \widetilde{\Omega_2} \wedge \\ & \mu_1 \text{ and } \mu_2 \text{ are compatible} \big\} \end{aligned}$$

with

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

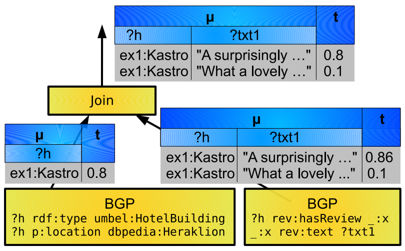
where tm *is an application-specific trust merge function.*

Again, the definition does not prescribe a specific trust merge function; thus giving applications a choice (insofar as the function is commutative and associative). Possible choices are the minimum or the arithmetic mean.

Example 3. The group graph pattern

{
 { ?h rdf:type umbel:HotelBuilding;
  p:location dbpedia:Heraklion. }
 { ?h rev:hasReview [rev:text ?txt1] }
}

groups two BGPs. Figure 7 depicts the corresponding algebra expression as an operator tree. The tree is annotated with trust weighted solution mappings that are solutions for our sample graph in Figure 2. The two tables near the bottom of Figure 7 contain the solutions for the BGPs. Joining the solutions from both multisets results in the solutions represented by the upper table. The trust merge function applied is the minimum function. □



μ		t
?h	?txt1	
ex1:Kastro	"A surprisingly …"	0.8
ex1:Kastro	"What a lovely …"	0.1

μ	t
?h	
ex1:Kastro	0.8

μ		t
?h	?txt1	
ex1:Kastro	"A surprisingly …"	0.86
ex1:Kastro	"What a lovely ..."	0.1

```
?h rdf:type umbel:HotelBuilding
?h p:location dbpedia:Heraklion
```

```
?h rev:hasReview _:x
_:x rev:text ?txt1
```

Fig. 7. Operator tree with solutions

Similiar to the join operator we adapted all algebra operators for tSPARQL [9].

5 SPARQL Extension for Trust Requirements

Section 3 gives a high-level overview of tSPARQL and the new clauses `TRUST AS` and `ENSURE TRUST`. In this section we provide a more formal description.

5.1 Accessing trust values

The `TRUST AS` clause permits access to the trust values associated with solutions. This is impossible with conventional SPARQL queries since conventional query evaluation does not consider trust values. To process `TRUST AS` clauses we extend the grammar of the query language, adapt the translation to an abstract syntax tree (AST) as well as the transformation from an AST to an abstract query; we define a new algebra operator and we extend the operation to evaluate abstract queries. Due to the limited space we only give a brief informal description of the grammar extension and present the algebra operator here. The tSPARQL specification [9] covers all necessary extensions in detail.

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. A `TRUST AS` clause can occur at any position in a query where `FILTER` clauses are permitted. The corresponding algebra operator, called *project trust operator*, operates on a multiset of trust weighted solution mappings. For every mapping it 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 8. *Let $\widetilde{\Omega}$ be a multiset of trust weighted solution mappings; let v be a query variable which is not bound in any $\widetilde{\mu} \in \widetilde{\Omega}$. The result of a* ***project trust operator*** *is a multiset of trust weighted solution mappings which is defined as*

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

with $\mathrm{card}_{PT(v,\widetilde{\Omega})}(\widetilde{\mu}) = \mathrm{card}_{\widetilde{\Omega}}(\widetilde{\mu})$.

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

Example 4. Consider a group graph pattern similar to the pattern in Example 3 where the second BGP is associated with a `TRUST AS` clause. Figure 8(a) depicts an operator tree, annotated with sample solutions, for this pattern. Compare the solutions consumed and provided by the project trust operator. Every solution provided by this operator contains an additional binding for variable `?t1`. This binding maps `?t1` to a value that corresponds to the trust value associated with the respective solution when the project trust operator is evaluated (e.g. 0.86 for the first solution). Note, we used the trust merge function $tm_{\min}$ for the join operation. Thus, the trust value of the first solution after the join is 0.8. However, the value bound to variable `?t1` has not changed; it is still 0.86. 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 where the `TRUST AS` clause has been defined for the whole group graph pattern (i.e. before the last closing brace in the pattern of Example 3). Figure 8(b) depicts the corresponding operator tree 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 `?t1` because the project trust operator is applied after joining the solutions. Obviously, the position of a `TRUST AS` clause in a query pattern matters. □

5.2 Expressing trust requirements

To express trust requirements as in query Q_2 we propose the `ENSURE TRUST` clause. Conventional `FILTER` clauses are not defined for trust weighted solution mappings and, thus, are inapplicable to filter solutions by restricting trust values directly. This holds for application-specific extension functions too. Instead of

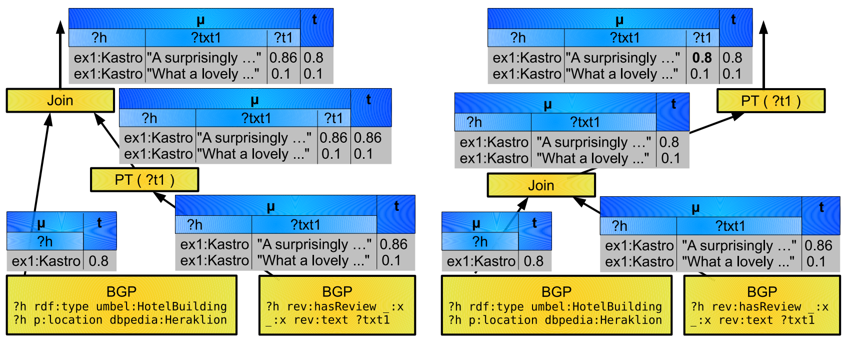


Fig. 8. Project trust operators in an operator tree with sample solutions

attempting to redefine `FILTER` we introduce the new clause as a clear separation between restricting with respect to variable bindings and associated trust values.

To process `ENSURE TRUST` clauses we extend the query language similar to our extensions for the `TRUST AS` clause. We refer to the tSPARQL specification [9] for the details and confine ourself here to present the new algebra operator.

Definition 9. *Let $l, u \in$[-1,1] be lower and upper bound values, respectively; let $\widetilde{\Omega}$ a multiset of trust weighted solution mappings. 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 ensure trust operator accepts only those trust weighted solution mappings that have a trust value within a specified interval, i.e., it eliminates any solutions with trust values lesser than the lower bound or larger than the upper bound. As for the `TRUST AS` clause the position of an `ENSURE TRUST` clause in a query pattern matters.

Please notice, the results of a tSPARQL query may differ for different users. As discussed in Section 2, trustworthiness of triples is a subjective judgment. Hence, the trust values associated with query solutions depend on the information consumer. For this reason, each tSPARQL query must be executed in the context of a specific consumer in order to determine consumer-specific results.

6 Optimization of tSPARQL Query Execution

A well-known heuristic to optimize query execution in relational database systems is selection push-down. Algebra expressions are being rewritten to push down selections in the operator tree in order to reduce 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 $tm_{\min}$.

Let $\widetilde{\Omega_1}$ and $\widetilde{\Omega_2}$ be multisets of trust weighted solution mappings. For join operators that employ $tm_{\min}$ the following equivalence of algebra terms holds:

$$ET\Big(l, u, Join\big(\widetilde{\Omega_1}, \widetilde{\Omega_2}\big)\Big) \equiv ET\Big(l, u, Join\big(ET(l, 1, \widetilde{\Omega_1}), ET(l, 1, \widetilde{\Omega_2})\big)\Big) \qquad (1)$$

Due space limitation we do not proof the equivalence. Instead, we refer to the tSPARQL specification [9] which contains proofs for all equivalences presented here. Based on (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:

$$ET\Big(l, u, LJoin\big(\widetilde{\Omega_1}, \widetilde{\Omega_2}, ex\big)\Big) \equiv ET\Big(l, u, LJoin\big(ET(l, 1, \widetilde{\Omega_1}), \widetilde{\Omega_2}, ex\big)\Big) \qquad (2)$$

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

$$ET\left(l, u, Filter(ex, \widetilde{\Omega})\right) \equiv Filter\left(ex, ET(l, u, \widetilde{\Omega})\right) \qquad (3)$$

$$ET\left(l, u, PT(v, \widetilde{\Omega})\right) \equiv PT\left(v, ET(l, u, \widetilde{\Omega})\right) \qquad (4)$$

$$ET\left(l_1, u_1, ET(l_2, u_2, \widetilde{\Omega})\right) \equiv ET\left(\max(l_1, l_2), \min(u_1, u_2), \widetilde{\Omega}\right) \qquad (5)$$

In contrast to (1) and (2), the equivalences (3) to (5) hold for all trust merge functions. Applying all the proposed rewrite rules during the optimization of tSPARQL queries reduces query execution times significantly (cf. Section 7.3).

7 Evaluation

In this section we evaluate the impact of our trust extension on query execution times. We implemented a tSPARQL query engine which extends the SPARQL query engine ARQ[2]. Our engine is available as Free Software from our project website[3]. For the evaluation we use a simple provenance-based trust function that assumes the existence of trust assessments for RDF graphs; these assessments associate a consumer-specific trust value with each graph. The trust function simply adopts these trust values for all triples in a graph. For our tests we use an extended version of the Berlin SPARQL Benchmark (BSBM) [10]. The BSBM executes a mix of 12 SPARQL queries over generated sets of RDF data; the datasets are scalable to different sizes based on a scaling factor. The generated data is created as a set of named graphs [11]. We extend the BSBM by trust assessments. Our extension, available from the project website, reads the named graphs-based datasets, generates a consumer-specific trust value for each named graph, and creates an assessments graph. The *assessments graph* is an additional RDF graph with statements that assign the generated trust values to the named graphs; for these statements we use our trust vocabulary (cf. Section 2). The proposed trust function determines the trust value for a triple by querying the assessments graph for the trust value associated with the graph that contains the triple. For all tests we use the minimum function to aggregate and to merge trust values. We conduct our experiments on a Intel Core 2 Duo processor with 2 GHz and 2 GB main memory. Our test system runs a recent 32 bit version

[2] http://jena.sourceforge.net/ARQ
[3] http://trdf.sourceforge.net/tsparql

of Gentoo Linux with Sun Java 1.6.0. In the following we, first, investigate how processing trust values impacts query execution time; we analyze the impact of determining trust values during query execution; and, finally, we evaluate the benefits of query rewriting as presented in the previous section.

7.1 The Impact of Processing Trust Values

As presented in Section 4.2 tSPARQL redefines the algebra of SPARQL in order to consider trust values during query execution. To measure the impact of this extension on query execution times we compare the results of executing our extended version of the BSBM with ARQ and with our tSPARQL query engine. To eliminate the effects of determining trust values in our engine we precompute the trust values for all triples in the queried dataset and store them in a cache. We execute the usual BSBM query mix for datasets generated with a scaling factor of 100, 200, 300, and 400; these datasets have the size of 31800, 60424, 92337, and 124305 triples, respectively. For each dataset we run the query mix 10 times for warm up and 50 times for the actual test. Figure 9(a) depicts the average times to execute the query mix with ARQ and with our engine, respectively. The main additional tasks performed by our engine, in contrast to ARQ, are accessing the trust value cache and aggregating trust values during BGP matching as well as merging trust values during join operations. Naturally, this additional functionality comes not for free. Nonetheless, the processing of trust values does not increase query execution times to a significant extent, especially for larger datasets, as can be seen in Figure 9(a).

7.2 The Impact of Determining Trust Values

While we analyze the processing of trust values in the previous experiment we focus on determining trust values during the execution of queries in the following. To measure how determining trust values may impact query execution times we use our tSPARQL query engine with a disabled trust value cache to execute the extended BSBM. During query execution the engine determines trust values with the simple, provenance-based trust function introduced before. For this setting,

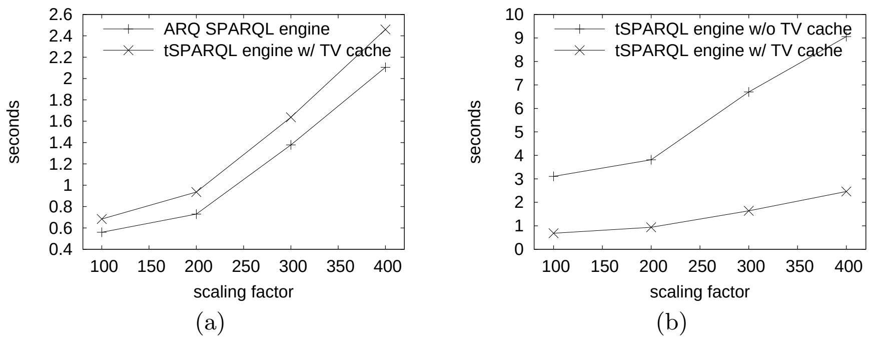


Fig. 9. Average times to execute the BSBM query mix for datasets of different sizes measured for ARQ and for the tSPARQL engine with and without a trust value cache.

the chart in Figure 9(b) depicts the average times to execute the BSBM query mix; furthermore, the chart puts the measures in relation to the times obtained for our engine with a trust value cache in the previous experiment. As the chart illustrates, determing trust values during query execution dominates the query execution time. Thus, we strongly suggest to make use of trust value caches in a tSPARQL engine. As a future work we will look deeper into the topic of caching trust values.

7.3 The Impact of Query Rewriting

In Section 6 we propose rewrite rules that push down trust constraints in order to execute tSPARQL queries more efficiently. To evaluate the potential impact of the application of these rules we implemented them in our tSPARQL query engine. Furthermore, we developed a mix of six tSPARQL queries. The queries are in line with the BSBM mix of SPARQL queries and with the BSBM e-commerce use case that considers products as well as offers and reviews for these products. Due to space limitations, we do not present our queries in detail; we refer the reader to the tSPARQL specification instead. However, to give an impression the following list shows two of the queries in their textual form:

– For all products of a specific type return the cheapest offer, respectively; ensure all information can at least moderately be trusted.
– Return reviews and the trustworthiness of these reviews for products of a specific type; ensure trustworthy information.

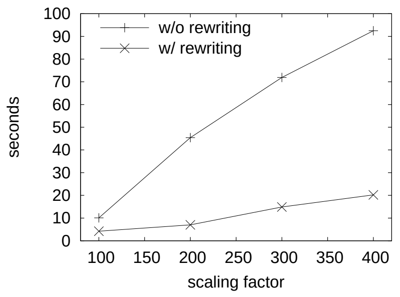


Fig. 10. The impact of rewrite rules

For our test we enable the trust value cache in our tSPARQL engine and compare query execution times when query rewriting is enabled and disabled. For both cases we run our query mix 10 times for warm up and 50 times for the actual test; we use four datasets (scaling factor: 100, 200, 300, and 400). As can be seen in Figure 10 the average execution times for a query mix differ drastically depending on the application of our rewrite rules. Hence, rewriting tSPARQL queries may reduce the execution time significantly.

8 Related work

Formalizing trust and trust in the Web is a topic of research since several years. For instance, Marsh [12] analyzes trust as a computational concept. This work influenced our approach to measure the trustworthiness of triples by a trust value in the interval [-1,1].

The basic idea of expressing trust requirements in a query language, as in our proposal, has also been presented by Bizer et al. [13]. They propose an extension to a query language for RDF, namely TriQL.P, to express trust-policies within queries. This approach is based on additional RDF statements that explicitly describe provenance and trustworthiness of information or information sources. Additionally TriQL.P permits a METRIC clause as "an open interface for different trust metrics and algorithms" which "are implemented as plug-ins for the query engine." In contrast to our approach, Bizer et al. do not explicitly provide a trust model for RDF data and their approach requires annotations regarding provenance and trustworthiness.

The most common approach to address trustworthiness on the Web are trust infrastructures based on a Web of trust (e.g. [1,2,3]). These Web of trust approaches consider the trustworthiness of members of the web. In contrast to these approaches, we focus on the trustworthiness of the data published on the Web, instead of the publishers. To the best of our knowledge, only Mazzieri and Richardson et al. propose trust models with an intention similar to ours. Mazzieri [14] introduces fuzzy RDF; a *membership value* associated with each triple represents the likelyhood the triple belongs to the RDF graph. By equating those membership values with trustworthiness of triples Mazzieri inappropriately mixes two different concepts; trustworthiness is not the same as a fuzzy notion of truth nor is trustworthiness of RDF triples tied to a specific RDF graph. Richardson et al.'s [15] approach is very close to ours; they represent a user's personal belief in a statement by a value in the interval [0,1]. What is missing in their approach is a possibility to finally utilize the ratings, e.g. in queries.

Other systems that consider trust in content are TRELLIS, IWTrust, and FilmTrust. The TRELLIS [6] system assesses the truth of statements by considering their provenance and related statements. Users can rate information sources and follow the assessments that are presented with the corresponding analysis and the influencing facts. The information assessed in TRELLIS is not as granular as the single triples in our approach and there is no trust model that rates the information directly. However, the presented possibility to assess information based on corresponding statements can be used to determine trust ratings of triples or sets of them. IWTrust [5], the trust component of the Inference Web answering engine, understands trust in answers as the trust in sources and in users. Similarily, FilmTrust [7] represents the trustworthiness of movie reviews by a user's trust in the reviewer and in other users' competence to recommend movies. These provenance-based notions of content trust may guide the development of trust functions.

9 Conclusion

In this paper we present a trust model for RDF data and tSPARQL, a trust-aware extension of the query language SPARQL. Our model associates every triple with a trust value. To access and use the trust values and to describe trust requirements we propose the TRUST AS and ENSURE TRUST clauses as extensions for tSPARQL. To enable tSPARQL we developed concepts for a trust-aware query processing. Conceptually, our solution is entirely independent from the applied methods to determine trust values; e.g. by not prescribing a specific trust function the two tasks, determining trust values and BGP matching, are clearly separated. Our approach can even be adapted for other trust models with a different definition of the trust values for triples; users merely have to provide trust functions and aggregation functions for their settings.

As future work, we plan to integrate tSPARQL in applications that process RDF data from the Web. Today, a majority of these applications do not consider the trustworthiness of the data. With tSPARQL we provide an easy tool to enhance these applications and make them more trust-aware. One of the requirements for integrating our concepts in applications is the existence of application-specific trust functions. For this reason, we will research different possibilities to determine trust values and we will integrate them in the tSPARQL query engine. In addition to trust functions applications require a method to aggregate trust values. We will develop trust aggregation functions and analyze their suitability for different applications and scenarios. Furthermore, we will develop concepts to enhance the trust value cache in our engine because efficient query execution benefits from caching trust values as our evaluation illustrates.

References

1. Golbeck, J., Parsia, B., Hendler, J.A.: Trust networks on the semantic web. In: Proc. of CIA2003. (August 2003)
2. Ziegler, C.N., Lausen, G.: Spreading activation models for trust propagation. In: Proc. of EEE2004. (March 2004)
3. Brondsema, D., Schamp, A.: Konfidi: Trust networks using PGP and RDF. In: Proc. of the Workshop on Models of Trust for the Web at WWW2006. (May 2006)
4. Hartig, O.: Provenance information in the web of data. In: Proc. of the Linked Data on the Web Workshop at WWW2009. (April 2009)
5. Zaihrayeu, I., da Silva, P.P., McGuinness, D.L.: IWTrust: Improving user trust in answers from the web. In: Proc. of iTrust2005. (May 2005)
6. Gil, Y., Ratnakar, V.: Trusting information sources one citizen at a time. In: Proc. of ISWC2002. (June 2002)
7. Golbeck, J., Hendler, J.: FilmTrust: Movie recommendations using trust in web-based social networks. In: Proc. of CCNC 2006. (January 2006)
8. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation (January 2008)
9. Hartig, O.: Specification for tSPARQL. http://trdf.sf.net/documents/tsparql.pdf (December 2008)
10. Bizer, C., Schultz, A.: Benchmarking the performance of storage systems that expose SPARQL endpoints. In: Proc. of the Workshop on Scalable Semantic Web Knowledge Base Systems at ISWC2008. (October 2008)
11. Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs, provenance and trust. In: Proc. of WWW2005. (May 2005)
12. Marsh, S.P.: Formalising Trust as a Computational Concept. PhD thesis, University of Stirling, Department of Mathematics and Computer Science (April 1994)
13. Bizer, C., Cyganiak, R., Gauss, T., Maresch, O.: The TriQL.P Browser: Filtering information using context-, content- and rating-based trust policies. In: Proc. of the Semantic Web and Policy Workshop at ISWC2005. (November 2005)
14. Mazzieri, M.: A fuzzy RDF semantics to represent trust metadata. In: Proc. of Italian Workshop on Sem. Web Applications and Perspectives. (December 2004)
15. Richardson, M., Agrawal, R., Domingos, P.: Trust management for the semantic web. In: Proc. of ISWC2003. (October 2003)