Trisolda: The Environment for Semantic Data Processing

Jiří Dokulil, Jakub Yaghob, Filip Zavoral
Charles University in Prague, Czech Republic
{dokulil, yaghob, zavoral}@ksi.mff.cuni.cz

Abstract

In order to support research and development of web semantization tools, methods and algorithms we have designed and implemented the Trisolda infrastructure. It is built around a semantic repository which is supplemented by import, query and data processing interfaces. The Trisolda application server can be extended by plug-ins for advanced semantic analysis and processing. We propose the TriQ RDF query language; its compositionallity and closedness make it useful for complex semantic querying.

Keywords: *Semantic web, infrastructure, repository, query languages*

1 Suffering of the Semantic Web

One of the main goals of the Semantic Web is to create a universal medium for the exchange of data. The Web can reach its full potential only if it becomes a place where data can be shared and processed by automated tools as well as by people. For the Web to scale, tomorrow's programs must be able to share and process data even when these programs have been designed totally independently [19].

Unfortunately, it seems, this goal has not yet been reached, albeit years of research by numerous researchers and large number of published standards by several standardization organizations.

We believe the Semantic Web is not yet widespread due to three prohibiting facts: missing standard infrastructure for Semantic Web operation, lack of interest from significant number of commercial subjects (although this started to improve recently), and the last but not least absence of usable interface for common users.

A nonexistence of a full-blown, working, high-performance Semantic Web infrastructure inhibits effective research of web semantization. Whereas the 'old web' has clearly defined infrastructure with many production-ready infrastructure implementations (e.g., Apache [20], IIS [21]), the Semantic Web has only experimental fragments of infrastructure with catastrophic scalability (e.g., Sesame [4], Jena [22]).

We have tried, during our experimental research, to convince commercial subjects to make somehow their data accessible on the Internet (of course with some reasonable level of security), and they all refused to make external access to their data. Commercial subjects do not intend to participate willingly in the ideas of the Semantic Web, because for them it either means to share their business data openly or to invest a lot of time and money for securing access to them.

Current standards in the Semantic Web area do not allow it to be used by common users. Whereas any user of WWW can easily navigate using hyperlinks in an available, production-quality WWW client, a contingent Semantic Web user has only a choice from a set of complicated query languages (e.g., SPARQL [13], SeRQL [3]). These query languages are not intended for casual users, only small number of people are able to use them.

Although SPARQL is probably the most popular RDF query language in the semantic web community, its overcomplicated definition and low expressive power make it unsuitable for most web semantization projects. Therefore we propose the TriQ query language that is based on the time proven ideas behind relational algebra and SQL.

The following chapters are organized as follows: after an overview of the infrastructure there is a description of the application server in Section 3 and the query API in Section 4. Sections 5 to 7 propose the TriQ language. Two final sections contain performance comparison and conclusions.

1.1 Related Work

Of course, the Trisolda infrastructure is not the only attempt to create an infrastructure for the Semantic web. One important example is the WSMX environment [15], which also represents a different approach to building the infrastructure. Unlike Trisolda, which is centered around the RDF database and Trisolda server, WSMX is concerned with semantic web services. It's purpose is to allow discovery (using Web Service Modeling Ontology [14]), mediation, invocation and interoperation of the services.

2 Infrastructure overview

We have recognized and described the problem of a missing, standard infrastructure for the Semantic Web in [17], where we have proposed a general ideas of a Semantic Web infrastructure and later refined the proposal in [7]. During the last year we have made a significant progress: we have implemented full-blown, working, fast, scalable infrastructure for the Semantic Web called Trisolda.

The figure 1 depicts the overall scheme of its infrastructure. In this picture rectangles represent processes, diamonds are protocols and interfaces, and grey barrels represent data-storages. All solid-line shapes depicts implemented parts of our infrastructure, whereas all dashed-line shapes represent possible experimental processes implemented by researchers playing with our infrastructure.

2.1 Trisolda repository

The heart of Trisolda infrastructure is a repository. It is responsible for storing incoming data, retrieving results for queries, and storing the used ontology. It consists of the a data-storage, which is responsible for holding semantic data in any format. Import interface enables fast, parallel data storing and hides details about background a data-storage import capabilities. The query interface has two tasks: to be independent on a query language or environment and to be independent on the Trisolda data-storage query capabilities. The last part of the repository is Trisolda Application Server. It is a background worker that does the inferencing, makes data unifications, and fulfills the task of a reasoner as well. It utilizes import and query interfaces for data manipulation.

2.2 Import paths

We use two distinguishable sources of data. The simplest one is a data import through importers from external data-storages. The task of importers is mapping external data-storage data-scheme to the SemWeb repository ontology. The second source of data crawls the wild Web using a web crawler; we have used Egothor [9] in the pilot implementation. The crawled web pages are stored in a Web pages data-store, where they can be accessed in parallel by deductors, which can deduce data and their ontologies from web pages and map them to our ontology.

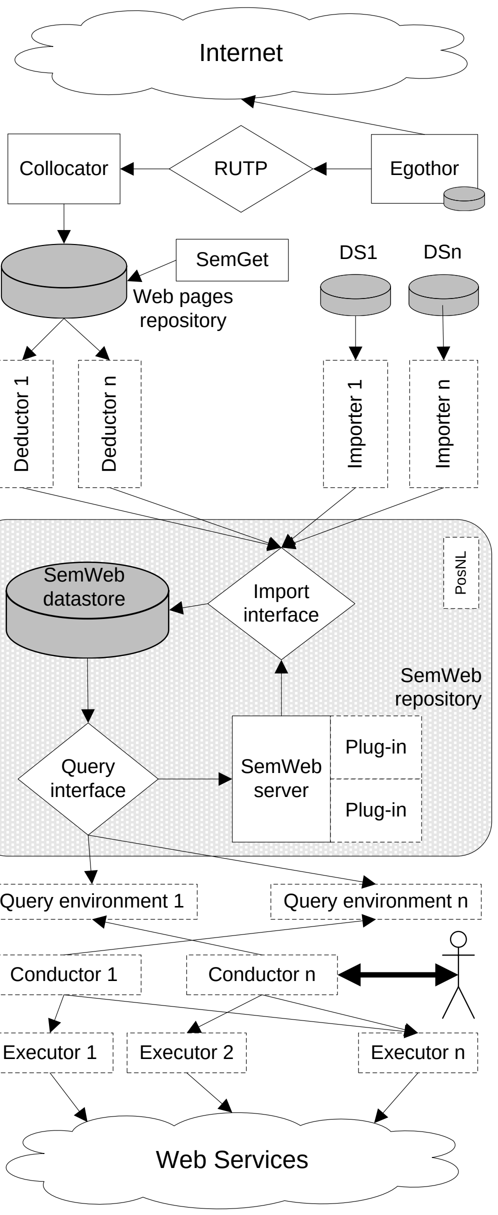


Figure 1. Infrastructure overview

2.3 Query environments

Query environments present outputs from Trisolda repository. They make queries using query API and present results to users in any feasible manner. We have implemented a SPARQL compiler as an example, which translates SPARQL queries to the internal query API requests.

2.4 Data-storage access

We have designed and implemented an object oriented library in C++ for a data-storage access independent on a background data-storage implementation. This library is used for a low-level access to the data in all interfaces to data-storages. It allows us to change an underlaying data-storage without modifying the code of our infrastructure.

2.5 Portability

Unlike other many research projects implemented usually in Java, we have decided to implemented nearly all parts (excluding Egothor implemented in Java independently on our project) in ISO/IEC 14882 C++. The main reasons are speed, more controlled computing environment (e.g., memory management), and, although it seems absurdly comparing to Java, stability.

When properly used, using ISO C++ brings full portability among different systems and compilers. Moreover, it allows us to implement bindings to other broadly used languages, e.g., Java or C#.

3. Trisolda Application Server

The main active part of the Trisolda repository is Trisolda Application Server. It is a background worker that does the inferencing, makes data unifications, and fulfills the task of a reasoner as well. It utilizes the import and query APIs for data manipulation.

It should be noted, that the server is not a web server in a conventional meaning. It does not handle any HTTP requests.

3.1. The server's role

We believe, we do not need to have all accurate data and inferences at the moment of data import. Just like the real world, the world knowledge changes at each moment and we are not able to catch it in one snapshot. Therefore postprocessing data in the background by Trisolda Application Server and computing some additional data in the background is acceptable and feasible.

The server is only a framework offering unified connection, interface and task management for experimental plug-ins, as described in the next sections.

3.2. Server's plug-ins

Trisolda Application Server's plug-ins are independent modules, which simultaneously perform different operations on SemWeb storage in the background. Whereas import and query APIs are only libraries enabling the access to the Trisolda storage, the server allows active operations upon the storage.

Each plug-in must conform to an interface requested by the server, whereas the server offers several classes of services for plug-ins.

3.3. Implementation

Plug-ins are implemented as dynamically loaded libraries. This is an important feature, which allows selective loading and unloading of any server's plug-in without interrupting overall infrastructure operation.

Although we use C++ to implement the whole project, the interface requested by Trisolda Application Server is in a C-like style, because there is currently no possibility to make a portable C++ dynamic library interface.

3.4. Executors

The results of querying semantic data can be interpreted by many methods. From relationally oriented data-set through set of references well-known by web search engines or set of mutually semantically related entities and their attributes up to application-level services using service oriented architectures.

Traditional result representation is tightly coupled to query method. SDE displays interconnected pages containing result data together with their structure and relationships, search engine displays web links with appropriate piece of text, and SPARQL returns rows of attribute tuples.

While many researchers are satisfied with making queries, the users (based on the ideas presented in [2]) would expect more from the Semantic Web. They expect it to take care of things, not just answer queries in a Google-like fashion.

The technique of executors brings process models into this infrastructure. The task of executor is to realize semantic action, i.e. interaction of result data with an outstanding (not only semantic) world. These atomic executors can be assembled to complex composed executors. Orchestration, i.e. mutual executor interconnection to achieve more complex functionality is executed by the Conductor module.

The technique of executors may be illustrated by following example. One's mother has gone ill, she needs a medicine. A query module searches nearby pharmacies with the medicine available. One executor is responsible for buying the medicine, while the other arranges delivery to mothers home. The Conductor orchestrates these two executors to synchronize, mutually cooperate, and pass relevant data between them.

3.5 Retrieving web documents

In the original proposal [17], there was a direct Egothor plugin for semantic data and metadata acquisition. This runtime structure would have several disadvantages.

The plugin dedicated to semantic experiments would run in the same environment as the general-purpose web robot. This would cause deficiencies in stability and performance. Moreover, it is harder to debug in this environment. Many semantic experiments need to apply several algorithms to a data set. Multiple data acquisition would cause unacceptable load to both the extractor and data providers. The web robot couldn't be dedicated to semantic data acquisition - it executes tasks for a lot of clients. Thus the delay between an initial request and document set completeness could be too long. We have decided to separate the web gathering and the semantic processing, both in time and space.

The retrieving document is converted into a stream of SAX events which enables us to process its internal structure more comfortably. This stream is then sent by a Robot UDP Transmitter Protocol (RUTP) [8] to a Collocator. The server reads document requests stored into the database, converts them into RUTP commands, sends them to the robot, receives streams of SAX events, completes them, computes tree indexes and in case of successful transmission stores each parsed document into the database.

The database stores each document in a structure similar to a XML Region Tree [11] or a NoK pattern tree [18]. The main feature of this structure is query effectivity - for a given element, all its ancestors or descendants in an element set can be queried with optimal cost.

4 Query API

The Query API is based on simple graph matching and relational algebra. Simple graph matching allows only one type of query. It consists of a set of RDF triples that contain variables. The result of the query is a set of possible variable mappings. This set can easily be interpreted as a relation with variable names used as a schema for the relation.

Relational algebra operations (e.g., joins or selection) are used on the relations created by simple graph matching. These operations are widely known from SQL, which was a major argument for this choice. Database developers are already familiar with these operations and a lot of work has been put into optimizing these operations.

So far, we decided to support only some of the common relational operations. Since the schema of elementary relations (results of basic graph patterns) consists of variable names, it is defined by the query and not in the database schema. For this reason, we use only natural joins. Variable names are used to determine which columns should the join operation operate on.

4.1 Selection

Selection operation revealed several problems specific to RDF querying. While in traditional relational algebra it is easy to maintain type information for each column, it is not possible in RDF. Even a simple query can produce a result that contains values with different data types in one column.

Having this in mind, we have to consider behavior of relational operators when it comes to different data types. For instance, in SPARQL [13] the operators should consider data types for each value separately, so one operator in one query compares some values lexicographically by their string value and some other values numerically by their value.

This is a serious performance problem that for instance makes it impossible to use indexes to evaluate expressions like $x < 5$ and especially $x < y$. On the other hand, such behavior is often not necessary because the user has certain idea about data type of x in $x < 5$. So we decided to make the type information part of the query. Then $x <_{integer} 5$ yields true for integers smaller than 5, false for integer greater or equal to 5 and error if x is not integer. This error always removes the whole row from the result.

This definition makes translation of queries to SQL more simple and efficient since it can be easily evaluated by a functional index that stores integral values. Conditions like $8 < x \ and \ x < 10$ can be evaluated by simply traversing a small part of this index.

4.2 Query language

We decided not to create yet another SQL-like query language. Since the query interface is intended to be used not by people but rather software, the query interface is actually a set of classes (an API). An example of a simple query tree:

- Natural join
  - Left natural join
    - Basic graph pattern P1
    - Basic graph pattern P2
  - Basic graph pattern P3

Had we created a query language, our form of query would basically be a derivation tree of a query in that language.

4.3 Query example

Following C++ code shows a simple query that outputs first and last name of all people that have both of them and whose last name is either "Tykal" or starts with "Dokulil".

```
Triples triples;
triples.push_back(Triple(
  Variable("x"), URI("http://example.org/lastn"),
  Variable("y")
));
triples.push_back(Triple(
  Variable("x"), URI("http://exmpl.org/firstn"),
  Variable("z")
));
Table *query_tab= new Filter(
    BasicGraph(triples),
    OrExpression( TestSubstringExpression(
        NodeExpression(Variable("y")),
        NodeExpression(Literal("Dokulil")), T, F
      ),
      EQExpression(
        NodeExpression(Variable("y")),
        NodeExpression(Literal("Tykal"))
  ) ) );

std::vector<Variable*> vars;
vars.push_back(new Variable("y"));
vars.push_back(new Variable("z"));
Query query(vars,query_tab,false);
```

The query consists of a basic graph query with two triples and three variables. Then a selection is applied to the result and finally a projection is used to return only columns with first and last name.

4.4 Query evaluation

We did not want to limit ourselves to just one system for data storage. Since the beginning of development we have been using four different data storages with several other in mind. Each of the systems offered different query capabilities from just evaluating all stored RDF triples to sophisticated query languages.

The contrast between a complex query API we wanted to give to the user and only basic query capabilities provided by the data storage system made it obvious that Trisolda must be capable of evaluating the queries itself. By implementing all operations within our system, we have reduced the requirements for the data storage engine to just one; the engine has to be able to list all stored triples. Thus the system is capable to use extremely simple storage engine that does nothing but read RDF triples from a Turtle file [1].

One of the other storage engines is an Oracle database. It would be highly inefficient to use the database only to list all triples, we want to utilize much of the Oracle optimized operations.

As a result, Trisolda is capable of evaluating any query itself, but tries to use any help the storage engine can provide. The same goes for adding new features to the query interface. Once the feature is implemented in our system, it is immediately available with all storage engines. Of course, performance of query evaluation will probably be suboptimal.

4.5 Evaluation algorithm

Since every storage engine can have specific capabilities, we could not establish a set of rules to decide what can be evaluated by the engine. Thus each engine contains an algorithm to determine whether it is able to evaluate a query. For query Q the evaluation plan is found like this:

- If the storage engine can evaluate Q then this evaluation plan is used.
- If Q is basic graph pattern then it is decomposed into individual triples, the evaluation plan for each triple is found and the results are joined together.
- If Q is an algebraic operation, evaluation plan for each operand is determined and the operation is applied to the results by Trisolda.

The limitation of this algorithm is, that it does not try to rearrange the query in order to achieve better performance either by choosing a more efficient evaluation plan or by allowing the storage engine to evaluate greater part of the query itself, which will probably be more efficient. This problem is a more complex version of optimization of relational algebra expressions and will be a subject of our further research.

4.6 Remote queries

Creating a data interface between different programming languages is not an easy task. To allow queries

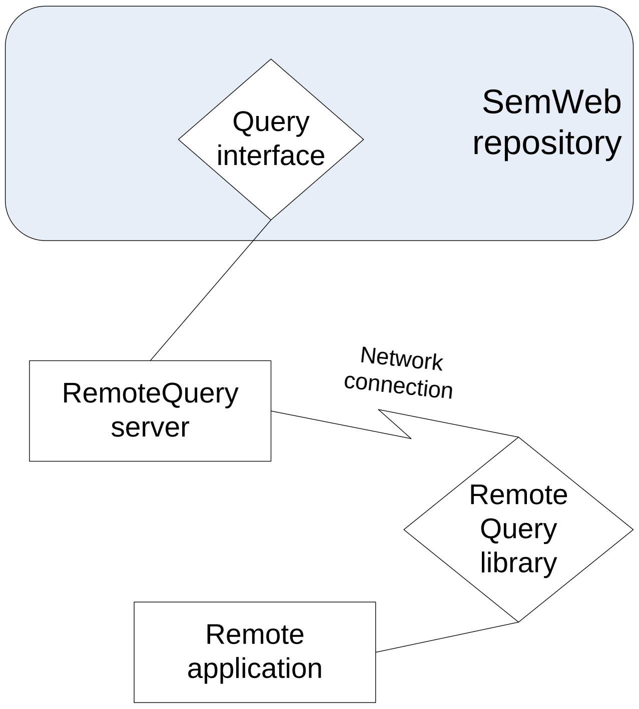


Figure 2. Remote query

from languages other than C++ we created a HTTP-based API that is language independent. The client issues a HTTP GET command that contains the query to be executed and receives a HTTP response that contains the results.

To make this possible, we had to create a way to send a query over HTTP. Since we do not use any textual query language, this required some work to be done. But due to a simple tree structure of the queries, serializing a query to a character string was a relatively easy task. The name of the class is stored in the string followed by the data. The data are either strings (stored directly) or other classes (the same serialization algorithm is recursively called to store them to the string).

An example of such encoded query could be this:

```
Query{0;BasicGraph{Triple{Variable{x};
URI{http://www.is.cuni.cz/stoh/schema/
ot_osoba#prijmeni};Literal{Dokulil;;}}
};x}
```

Furthermore, the format of the results has to be decided. To make the server as fast as possible the in-memory data format used internally by the server is used. The format is suitable for transfer over network to different platforms - it has no little/big endian or 32/64 bit compatibility issues and it only uses small amount of information other then the actual data. Only 5 bytes plus size of a URI (encoded as UTF-8) is required to transfer the URI, 5 bytes plus length for untyped literal, 9 bytes plus length of value and type for typed literal, . . .

We have implemented and successfully tested a C# client library.

4.7 Complex query languages

One of the ways in which the query API can be used, is to build more sophisticated query languages. An example is a limited SPARQL [13] evaluator or the Tequila query language [10]. The languages represent very different approach to RDF querying but they both use the same API and thus access the same data.

The Tequila language is based on named patterns and supports recursive queries. An interesting example is a selecting employees from a list. The lists in RDF are recursive, which makes the following query impossible in SPARQL.

```
prefix rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ex: <http://www.example.org/term#>

ex:list(?N)
{
  {
    ?N rdf:first ?F.
    ?N rdf:rest ?R.
    use ex:list(?R)
  }
  union
  {
    filter ?N = rdf:nil.
  }
}
get
{
  ex:department ex:employees ?list.
  use ex:list(?list)
}
```

The ex:list named pattern has one parameter and is used recursively to traverse the whole list.

Although this is only a very basic example, it clearly demonstrates the possible diversity of query languages that can be build over Trisolda.

5 TriQ

Querying is one of the important issues for any data format. In the case of RDF, many query languages have been developed, including the SPARQL language

[13], which is a W3C recommendation, or SeRQL [3], which is supported by the popular Sesame RDF framework [4].

Many are inspired by SQL (although not all of them, e.g., Lisp-like Versa [12]). But the inspiration is usually manifested (besides the fact that the queries syntax may look a bit like SQL) in the fact that the RDF graph is transformed by some graph pattern matching operation into some table-like form and these tables are then further processed. We believe that the inspiration should have been a little different. An important feature of SQL (and its theoretical background – the relational algebra [5]) is the fact, that it is a closed system. Relations are transformed into relations. This way, a result of a query can be used as an input for another, more complex query, which is impossible in SPARQL and SeRQL.

The following parts of this paper present our proposal for TriQ – a SQL-inspired, closed RDF query system. To make the system closed, we couldn't have used relations as our data model. We use RDF. But the operations are inspired by relational algebra – we use selection, projection, (inner and outer) joins, etc. The semantics of these operations is not exactly the same (after all, we have a very different data model) but the ideas behind them are the same. We believe that this (and the fact that it is closed) makes the whole query system more accessible for wider audience of developers, especially those with long SQL experience.

5.1 Data model

Since we want a closed query system, we require every operation to take some RDF graphs (zero or more) as its input and produce an RDF graph as its output. But to make the operations simple to use, we need to add some further information. We use the very RDF that contains the data for the task and add additional triples to the data – *decorate* it.

Namespaces To make decoration simple we define several namespaces. URIs starting with theses namespaces are prohibited in the queried data. The namespaces are
`dec` is `http://ulita.ms.mff.cuni.cz/Trisolda/GQL/decoration`
`ptr` is `http://ulita.ms.mff.cuni.cz/Trisolda/GQL/pointer`
`graph` is `http://ulita.ms.mff.cuni.cz/Trisolda/GQL/graph`

5.2 Decoration of nodes and edges

We can decorate either nodes of the RDF graph, in which case we add a new triple where the decorated node is the object of the triple, or edges, in which case we have to reify the edge (unless the triple is already reified) and then decorate the reification. To be more specific, to decorate the edge `S P O` with decoration triple D_S D_P `?` (the question mark is the decorated object) we add the following triples:

- `X dec:subject S`, `X dec:predicate P`, `X dec:object O`
- D_S D_P `X`

The X denotes an anonymous node. Note that we do not use the standard reification defined by RDF, but rather use the `dec` namespace to avoid potential "collisions". This way we can always distinguish statements added during decoration and statements that were present in the original data.

Decoration options There are two types of decoration edges. Let X be the decorated object (either a node or reification of an edge), G an URI from the namespace `graph` (each graph has a globally unique URI) and P an URI from the namespace `ptr`. The possible decoration triples are:

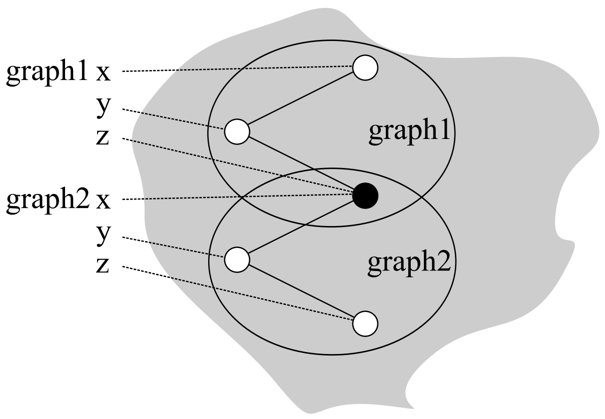
1. `G dec:contains X`
2. `P G X`

There are no restrictions for the second type of triples. The only restriction for the first type is that every node and edge of the decorated graph is decorated by at least one such edge.

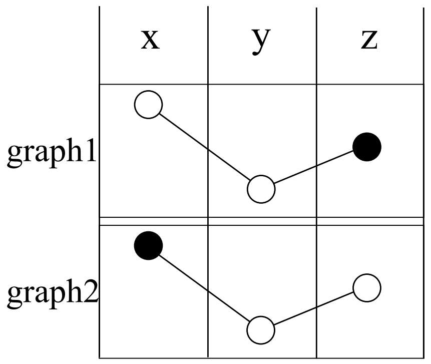
5.3 Meaning of decoration

The purpose of decoration is to help user define a structure in the queried graph and exploit it to define further operations. Furthermore, we would like the whole query system to resemble relational algebra, that works with relations – sets of tuples with a well defined schema.

The first type of decoration triples is used to make the (one) RDF graph appear as if it was a (multi)set of smaller graphs so that each of the smaller graphs resembles one tuple of a relation (row of a table). The triple `G dec:contains X` tells us that the graph G contains X. So if we take all such X for one G, we get one small RDF graph.



(a) Multiple graphs in one graph



(b) Table analogy

Figure 3. Meaning of decoration

The second type of decoration triples were introduced as a parallel to schema of a relation (for an example see the Figure 3). The set of all values from the namespace `ptr` used in the graph are schema of the graph. All values of X from all triples `P G X` for a graph G correspond to a value of column P in a row G of a table. But unlike SQL that allows zero (NULL) or one value, we allow zero or more values. Although we could restrict it to just one value, we believe it would unnecessarily limit the graph-handling capabilities of the query system. For instance, we would like to be able to create such set of graphs, where each graph contains information about one person (e.g., first and last name) and all of his or her e-mails. And to make handling of the data convenient, we would like to have a pointer ptr:first-name point to the first name of the person, ptr:last-name to the last name and ptr:mail to all of the emails. That way, we could for example easily find people with more than one email or get the number of emails for each person.

6 Graph pattern operation

The graph pattern operation is in a certain sense the very basic operation of the query system. It is used to find patterns in the whole queried data. The very basic principle is that the operation specifies an RDF graph where some nodes or edges are replaced by variables. The evaluation is done by finding possible substitutions for these variables so that the we get a graph that is a subgraph of the queried data. This graph is then one member of the result set.

The operation gives the data a structure that helps us reference certain concepts in further query operations. For example, if the pattern looks like `?x ex:has-name ?y` (where ?x and ?y are variables), we know that the actual nodes bound to ?x are people and ?y their respective names (provided we have reasonable data).

Each node and edge can of the pattern can have a pointer assigned to it. In that case, the corresponding node or edge of the result is then pointed to by that pointer.

Why? Many RDF query languages (e.g., SPARQL [13], Trisolda query API [7],. . .) use some kind of graph patterns to transform the RDF graph to a table (or something analogous like set of variable mappings in the case of SPARQL). In other words, they use it to transform the queried data into some other form suitable for further processing.

At the moment, we believe this is the only operation that should work with "raw" data and that all other operations can assume to be working with decorated data. This is not as important from the technical or formal point of view, but rather from the "average user's" point of view. It would allow him or her to construct the query in two steps. First, well structured pieces of data are defined by the pattern operation. Second, the structure is exploited to combine the pieces of data into the final result. The second phase should be as close as possible to writing a query in SQL.

6.1 Definition

The previous sections briefly and informally explained what capabilities the proposed pattern matching possesses. This section gives a more formal view of the operation.

Let Uri be a set of all URIs, Lit set of all RDF literals, $Blank$ set of all blank nodes, and Var an infinite set of variables.

Let $V \subseteq Uri \cup Lit \cup Var$. The pattern P is then defined as a non-empty set of triples $P \subseteq V \times (Uri \cup Var) \times V$. V is the set of all nodes in the pattern P, $Var(P)$ denotes set of all variables used in the pattern P. The pattern can be viewed as a directed, labeled multigraph. We require that each two nodes in V are connected by an undirected path.

Each edge or node of the patter can be assigned a pointer (i.e. URI from the namespace `ptr`). The same pointer may be assigned to more objects (edges and nodes).

Let P_b be a pattern and V_b nodes of P_b. Let G be the queried data.

A variable mapping is a function $\mu : Var \rightarrow Uri \cup Lit \cup Blank$. We extend the variable to triples $t \in P_b$ and the whole pattern P_b in the natural way (each variable v used in t or the whole P_b are replaced by $\mu(v)$). We always use the variable mapping μ in conjunction with a pattern P, in which case we consider only minimal mapping, i.e. $dom(\mu) = Var(P)$.

We say, that an RDF graph M_b is a *match* for P_b iff there are mappings μ and η such that the following statement holds: $M_b = \mu(P_r) \Subset G$ where $\Subset$ is a relation between RDF graphs. The basic version of TriQ assumes that $(A \Subset B) \equiv (A \subseteq B)$.

Then we say that the tuple $\langle \mu, M_b \rangle$ is a *result* for the pattern P_b. Note that we only consider minimal μ mappings, i.e. mappings that only map variables used in P_b. The M_b graph still has to be properly decorated according to the pointers that were assigned to the pattern P_b. We add decorating triples to the set M_b in the way described in the Chapter 5.1 – if a pointer was assigned to a node or edge of the pattern, we add the appropriate decoration to its image under $\mu(P_b)$.

7 Algebraic operations

This section describes algebraic operations, that run on the decorated data and further filter and transform it. Although, strictly speaking, these operations could be run on undecorated data, but there is usually no reason to do so. In such case, each node and edge of the undecorated data would be decorated by the first type of decoration triples, which would assign the whole data to one graph.

We do not give formal definitions for these operation as they are quite straightforward and usually obvious. They would only add a few pages of not very interesting technicalities to the paper.

7.1 Selection

The selection operation has one argument and tests a condition for each graph g in the argument multiset. If the condition is true, the graph is added to the result. The basic idea is the same as in relational algebra, but there is some added complexity due to the fact that one pointer can have more than one value (within one graph) or no value at all. We use a language derived from the first-order predicate calculus to construct the expressions. The main difference from SQL is the addition of quantifiers. The quantifiers are always in the form $Q_{x \in X}$ where x is a variable, X a pointer from the schema of the operand and Q either $\forall$ or $\exists$. The rest of the expression is formed from the variables, functions, predicates and logical operators. There are some limitations. One variable cannot be used in more than one quantifier and whole expression must be closed (meaning that there is no unquantified variable and no variable is used outside of the range of the quantifier for that variable). $\forall_{x \in X}$ denotes that the quantified condition must be true for each x from $PtrVal(X, g)$ and $\exists_{x \in X}$ denotes that there mast be at least one x in $PtrVal(X, g)$ such that the quantified condition is true. $PtrVal(X, g)$ denotes values of all nodes pointed to by X in the graph g and predicates of all edges pointed to by X in the graph g.

Example: $(\exists_{p \in ptr:Payment}(p > 1000)) \wedge (\forall_{r \in ptr:Person} \exists_{c \in ptr:Customer}(r = c))$. This means that the graph must have a value p for variable $Payment$ that is more than 1000 and that for each value of $Person$ there is the same value for $Customer$.

The formal definition is very strict, but the actual query language can be more relaxed, allowing the user to write less verbose queries as long as there is a clearly defined transformation to the form defined here.

We do not attempt to list all functions and predicates. In general, we assume that there is always a (hidden) parameter that carries the currently processed graph as its value – in the strict definition of the model it is a pair containing the whole graph and subset identifier from the namespace `graph`.

Some of the functions and predicates we would like to include are:

- $\textsc{PathLength}(x, y)$ that return length of a path between nodes x and y.
- $\textsc{Sum}(A)$, $\textsc{Max}(A)$, $\textsc{Min}(A)$, $\textsc{Count}(A)$ that return the sum, maximum, minimum or number of nodes that the pointer A points to.
- $\textsc{IsURI}(x)$, $\textsc{IsLiteral}(x)$ that check, whether the value of x is of the specified type.

- TypeOf(x) that returns the data type of the literal x.

A detailed proposal of the language would include several basic functions and data type conversion rules as well as extension mechanism that would allow implementations to add further functions.

7.2 Projection

The purpose of the projection operation is to remove unneeded parts from the data. It has one argument and the operation is specified by a set S of URIs from the namespace `ptr`. Any reasonable set S should be a subset of the schema of the argument, but it is not required. The operation removes triples from its argument. There are two versions – induced and non-induced.

The non-induced version removes all non-decoration triples that are not pointed to by a member of S and all decoration triples that represent pointers not in S. Then all decoration triples g `dec:contains` o (graph membership) are removed if there is no triple $p\ g\ o$ where $p \in S$. Note that if we remove a decoration triple that decorated an edge, we remove the three reification triples as well.

The induced works the same as non-induced except that it does not remove edges, where both endpoints are being pointed to by members of S.

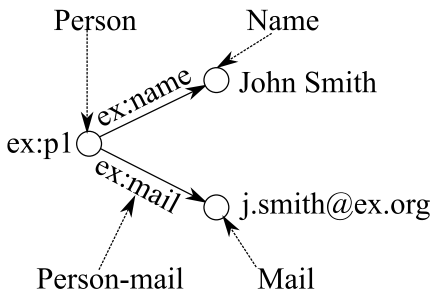
The Figure 4 gives an example where $S = \{Person, Mail\}$.

7.3 Distinct

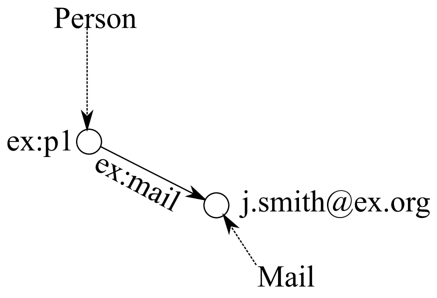
So far, each operation generated a multiset of graphs. The distinct operation takes one argument and eliminates all duplicates. The equivalence of the graphs does consider decoration as well, i.e. for two graphs to be equal, even the pointers in both graph must point to equal nodes and edges.

7.4 Joins

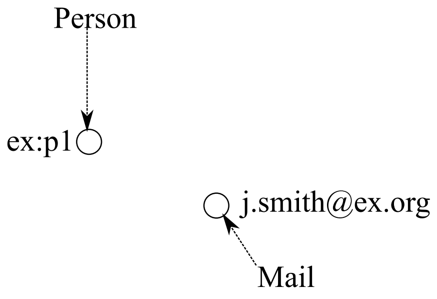
Joins are an important part of relational algebra and SQL. As we are trying to get close to these languages, we also introduce join operations. Join is a binary operation that produces results by making a Cartesian product of the arguments and then filters the results according to a condition. There are special variants of the operation – outer joins (left, right and full). The basic (inner) join could be defined as a combination of cross join (i.e. Cartesian product) and selection, but the outer joins are more complex so we have decided to include “whole” join operation.



(a) Input



(b) Induced projection



(c) Non-induced projection

Figure 4. Projection

The join can be seen as an operation that generates one small RDF graph for each pair of graphs where one is from the first argument and the other from the second argument. The graphs are union-ed together and if the produced graphs fulfills the join condition, it is added to the result.

The outer joins work just like in SQL. Consider for example left join. If there is a graph l in the left argument such that there is no graph r in the right argument that $l \cup r$ fulfill the join condition, then l is added to the result.

An example of a left join is in the Figure 5. The left and right operands are joined by a left join on a condition $Person1 = Person2$ (of course, the actual condition should contain the appropriate quantifiers).

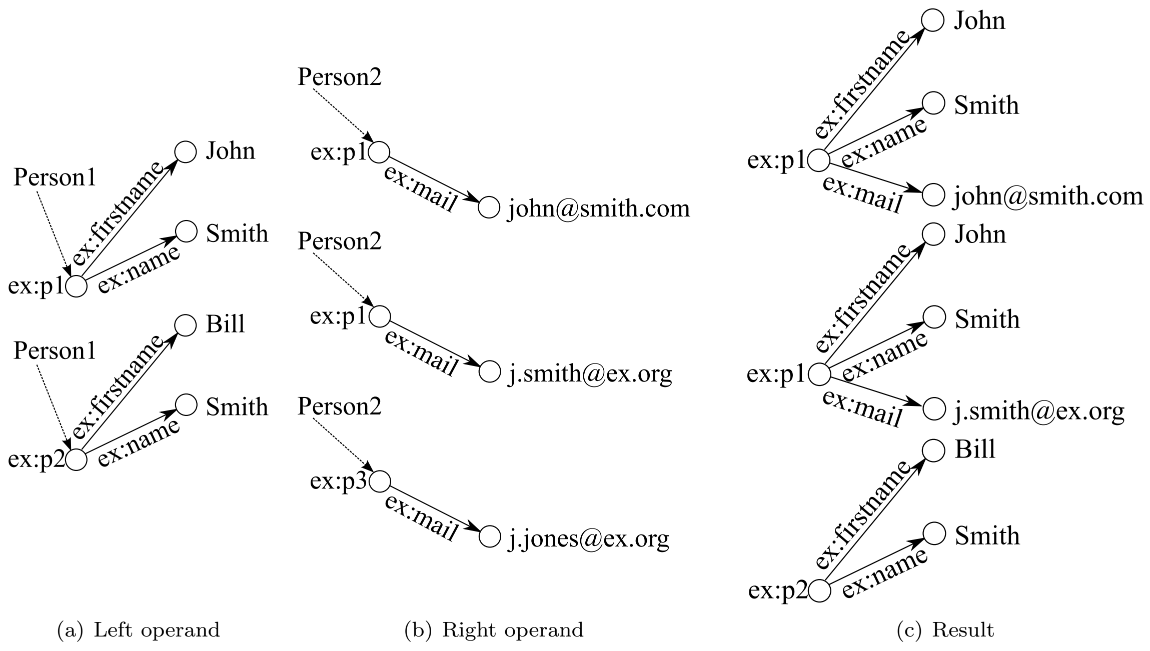


Figure 5. Left join of names and e-mails

7.5 Group by

In SQL, the "group by" construct is almost exclusively used with aggregation. But as our data model allows more values per "row", we can use "group by" as a standalone operation that groups related data together. To be more specific, it joins (makes a union) graph, that have the exactly the same values for a specified set of pointers.

The Figure 6 shows an example where two graphs are grouped by the value of the "Person" pointer into one graph.

7.6 Aggregation

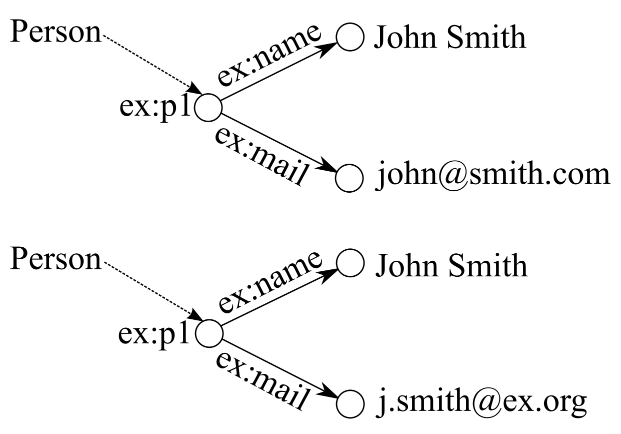
A very important feature in SQL are aggregation functions. An important difference between our data model and SQL is that a pointer can point to more than one value within each graph. So there are two possibilities, where aggregation functions can be used. They can either aggregate values within one graph or make aggregations over whole data. We decided to allow both. A *local* aggregation has the form $fnc(ptr) \rightarrow res$ where fnc is an aggregation function (min, max, sum, count, avg and "distinct" variants of sum, count and avg), ptr is a pointer and res is a pointer. The aggregation function is evaluated for each graph g and for a result v, the triples `g gql:contains v` and `res g v` are added to the data.

The *global* aggregation has the form $fnc_g(fnc_l((ptr)) \rightarrow res$ where fnc_g and fnc_l are from the same set of functions as in the local variant. The function fnc_l is used to compute aggregation over each graph and then fnc_g combines these results into one final value v. The result contains only one graph g with triples `g gql:contains v` and `res g v`. Because the data are "destroyed" more global aggregations can be specified in one aggregation operation.

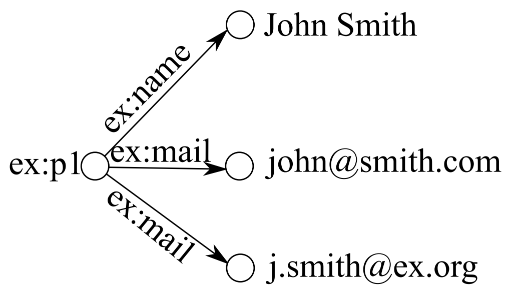
An example that demonstrates why we decided to define global aggregation like this is the following. Consider the already familiar data about people and e-mails. We can use a global aggregation $max(count(Mail)) \rightarrow MaxMail$ to get the maximal number of e-mails the people have. Then, on the same source data, we run a local aggregation $count(Mail) \rightarrow MailCount$. Then we join the data on $MaxMail = MailCount$ to get information about everyone with maximal number of e-mails. Note, that since we included the aggregation functions among the function that can be used in the selection operations, we could omit the local aggregation step in the example and use $MaxMail = \text{COUNT}(Mail)$ as the join condition.

7.7 Set operations

Some set operations are also present in SQL – union, union all, intersect, and minus. The equivalent of union



(a) Input



(b) Grouped data

Figure 6. Group by

all is obvious, it simply returns union of the two graphs (since we require that graphs are identified by globally unique identifiers, there can be no "collision"). The is no such natural equivalent for the other three operations. The problem is that in our data model, "schema" is much more relaxed concept than in SQL – multiple values of a pointer, nodes and edges without pointers or with several different pointers, etc. Should the set operations be performed only with the data in graphs without regard to pointers (and what would be the pointers of the result) or with pointers? Perhaps it would be best to let the user specify a set of pointers and the operations only consider nodes and edges with these pointers.

But this creates a completely new problem. If we make an intersection of two sets and each of them contains a graph that is different, but the values pointed to by the specified set of pointers are the same. What should get into the result? The first or the second? That would make the intersection operation non-commutative.

We decided to use the following definition for the operations (A and B are operands, S is a set of pointers, $g[S]$ denotes a projection of graph g to the set of pointers S):

- A *union*$_S$ B is a shortcut for grouping operation with columns S applied to A *unionall* B
- A *minus* B are all graphs g of A such that there is no graph g' in B for which $g[S] = g'[S]$ holds. Projection $g[S]$ is either induced or non-induced – the exact version is specified by the user.
- A *intersect* B is not included as an operation. Although we came up with several possible semantics, none of them seemed more natural than the others. This and the fact that they could all be transformed into some combination of other operations led us to the decision not to include any of them as a build-in operation.

7.8 Constructors

Transformation from one RDF graph to a different one is a big problem for all RDF query languages. Many of them contain some kind of CONSTRUCT concept – a graph pattern is specified and new RDF graphs are generating by substituting each row of the query result into the pattern. Since we have no rows in the result (only an analogy that cannot help us in this case), we cannot use this approach.

Such operation would be extremely useful addition to our query system, since we could use it anywhere in the query and immediately perform other operations on the transformed data. Unfortunately, we have yet to find a simple and convincing definition for the operation. It is one of our immediate goals, perhaps the most important one.

7.9 Implementation concerns

We have defined operations of a RDF query system. We have not defined a query language, nor are we going to do so in this section. However important it may seem, it is in fact only a technical problem of defining a suitable textual representation for the presented operations. It has to be done and it has to be done carefully, since a bad language with complex grammar that makes it unclear and unreadable would certainly discourage developers from using the whole query system, no matter what the underlying operations and data model are.

From the database point of view, the implementation of the query system would probably be more complex than in the case of the table-oriented RDF query languages, that can often be easily transformed into SQL queries over some representation of the RDF triples in a relational database. Although there are some significant performance issues involved, they can be greatly reduced. And the huge amount of work and money that have been spent on improving reliability and performance of RDBMS products are a great advantage of such solutions.

Although storage and transaction handling for an implementation could most likely be built on top of some existing solution, query processing and optimization will have to be written from scratch.

8 Performance tests

We have made three sets of tests. The first one was a comparison between load time into one of existing RDF repositories based on a relational database and Trisolda data store that is also based on relational database. The second one was designed to predict the load time curve for large semantic data and the last one compared query times between Trisolda RDBMS-based and non-RDBMS-based data stores. Tests used different data described in Table 1.

8.1 Test environment

The test environment consist of two machines. The first one hosts a Oracle db server (2xCPU Xeon 3.06 GHz, DB instance was assigned 1.0 GB RAM) and the second one is an application server (2xCPU Quad-Core Xeon 1.6 GHz, 8GB RAM).

All tests used relatively large data containing 2.365.479 triples (303 MB Turtle [1] file).

Name	Description
DATASET_1	2.365.479 triples, 184.461 URIs, 53.997 literals, 303 MB Turtle [1] file
DATASET_2	26.813.044 triples, 2.020.212 URIs, 1.043.337 literals, 3396 MB Turtle file

Table 1. The data used in tests.

8.2 Data import

The main goal of this test was to compare Trisolda data store with an existing solution based on a relational database. As an example of an existing Semantic Web data store was chose the Sesame v1.2 due to its popularity in the Semantic web community. New version of Sesame (Sesame v2.0) doesn't support relational databases. Both Sesame-db and Trisolda data store were connected to a local instance of Oracle database.

We tried to load 150 000 triples DATASET_1 into both of them. The Trisolda data store loads this data in 780 seconds. The Sesame-db finished loading near 118 000 loaded triples and failed with a database error. The error reported was low space in the TEMP tablespace.

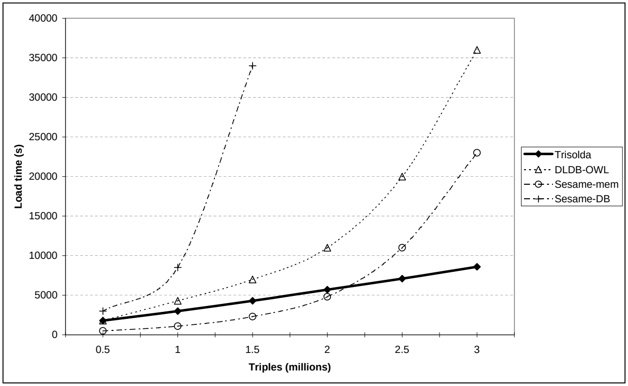


Figure 7. Data import comparison

Load times for the Sesame-db and the Trisolda data store are shown on Figure 7. The load time of Trisolda data store has almost linear dependency on the size of the processed data, but the Sesame-db exhibits rather exponential growth. The behavior of Sesame-db is expected and it is the same as described in [16].

One of the major design goals of Trisolda was storing huge semantic data. On the other hand, the Sesame database schema and SQL statements are not very suitable for loading huge data.

According to the test, smaller data (up to 110 000 triples in the machine configuration we used) may be loaded in Sesame-db, but it is not suitable to use the Sesame-db for larger data.

8.3 Huge data load

The main goal of this test was to determine whether the Trisolda data store is capable of loading huge RDF data (DATASET_2). During the implementation, we tried to identify possible bottlenecks and were able to eliminate some of them. There are still some performance issues; it is one of the subjects of our further work.

The data was loaded in 100k triples batches. Whole load took 22 hours and 54 minutes, out of which 13 hours and 44 minutes were spent transferring data from source data file to temporary tables in the database and another 30 minutes were spent on cleanup actions.

8.4 Query performance

Although we have tried to implement the algorithms used in query evaluation in an efficient manner the algorithms themselves are only basic versions so the performance of the query evaluation leaves a lot of space for improvement.

We have tested three storage engines: BerkeleyDB based storage that stores triples in a B-tree, fully in-memory engine, Oracle-based RDF storage.

First, we measured the performance of evaluation of the query presented in section 4.3.

The BerkeleyDB-based storage engine required 1.8 seconds to complete the query, while in-memory engine took only 0.7 seconds. The performance of Oracle-based engine was the worst, requiring 6.4 seconds.

We have expected these results. The current in-memory engine is read-only and is optimized for best performance in queries similar to the one we tested. On the other hand, we used the Oracle database only to provide us with plain RDF triples and performed the join operations in our system. But this is not the main reason for the bad performance. The problem is, that the Oracle database is placed on another server and network delays for each returned triple add together. Had we used the Oracle database to join and filter the results the performance would have been much better due to smaller network trafic and better optimization of joins in Oracle. Our measurements showed that time required to evaluate this query is around 0.2 seconds.

8.5 Oracle query performance

We have performed several performance tests over our Oracle-based RDF store. The queries were completely translated to SQL and then evaluated by the Oracle server. An example of a very basic query (basic graph pattern with one triple) looks like this:

```
SELECT x_l.lit_rec_type AS x_kind,
      x_l.lit_value AS x_value,
      (SELECT lng_value
         FROM adt_lang
        WHERE lng_id = x_l.lit_lang_id)
        AS x_lang,
       (SELECT dtp_value
          FROM adt_data_type
         WHERE dtp_id = x_l.lit_type_id)
       AS x_type
 FROM (SELECT x
   FROM (SELECT tri_subject_lit_id AS x
     FROM dat_triple t, dat_literal s,
          dat_uri p, dat_literal o
     WHERE t.tri_subject_lit_id=s.lit_id
       AND t.tri_object_lit_id=o.lit_id
       AND t.tri_predicate_uri_id
           = p.uri_id
       AND tri_predicate_uri_id =
          (SELECT uri_id
             FROM dat_uri
            WHERE uri_value
                  = :p1_predicate)
       AND tri_object_lit_id =
           (SELECT lit_id
              FROM dat_literal
             WHERE lit_value
                   = :p1_object))) q
       LEFT JOIN dat_literal x_l
              ON q.x = x_l.lit_id
```

In the following text, some queries are said to complete *instantaneously*. This means, that their evaluation time was comparable to the network latency (the database resides on a different server).

The queries in the following text are written as triples, where ?x denotes variable x, $<uri_1>$ denotes a URI with a value 'uri' and "value" denotes literal with value 'value'. The actual values are not given, as they are rather long and would be meaningless to the reader without deeper knowledge about the data used in the experiment.

The first query consists of basic graph pattern with one triple in the form ?x, $<uri>$, "literal". This query returned 10 rows and evaluated instantaneously.

The second query contained two triples: ?x $<uri_1>$ "literal1", ?x $<uri_2>$ "literal2". The query evaluated instantaneously and returned one row.

The next query was ?x ?y "literal". This query required 8 seconds to evaluate and returned 4 rows. On the other hand, the query $<uri>$?x ?y evaluated instantaneously returning 28 rows.

A more complex query ?x $<uri_1>$ "literal1", ?y $<uri_2>$?x, ?y $<uri_3>$?z, ?y $<uri_4>$ "literal2", ?y $<uri_5>$?w, ?w $<uri_6>$ "literal3" that returned only one row took as much as 200 seconds to evaluate. With the knowledge about the structure of the data, one could easily come up with an evaluation plan that would evaluate (nearly) instantaneously. But due to the way that data are stored in the database, the statistics that the Oracle server utilizes are unable to provide this. Dealing with this problem will be one of the subjects of our future research.

All queries presented so far only returned small result sets. We also measured one query ?x <uri_1> ?y1, ?x <uri_2> ?y2, ?x <uri_3> ?y3 that returned 88964 rows. This took 70 seconds.

Another 'big' query was ?x <uri_1> ?y, ?z <uri_2> ?x, z<uri_3> ?w and produced 184179 rows in 66 seconds.

The main reason for relatively long evaluation times is not caused by transferring the results from the Oracle database over the network. This transfer is just a matter of seconds even for the largest result set. Most of the time was spent on the actual evaluation of the query by the Oracle database.

The experiments have shown, that queries like "give me first and last names of all people in the database" are much slower than what they would be if the data was stored in a traditional relational database. The fact that each triple is stored separately and table join has to be performed is one obvious factor. Less obvious but just as important is the fact that the statistics used by the Oracle optimizer to create query evaluation plans do not work well if the data is stored like this (all triples are stored in one table) and the optimizer makes wrong assumptions. This means, that the optimizer works with inaccurate estimations of the size of data at most places of the evaluation tree. This makes the optimizer select wrong order of the joins and also inefficient methods (like using nested loops to join large relations). The problems are very similar to those identified in [6].

9 Conclusion

We have implemented and thoroughly tested the infrastructure for gathering, storing and querying semantic data. We have focused our efforts on efficiency, extensibility, scalability and platform independence. Both our experiences and benchmarks show that this goal is feasible.

Trisolda is currently used as a platform for further web semantization research. We expect to enhance both interfaces and functionality to support these semantic experiments.

Our immediate goal is to implement the TriQ evaluator within the Trisolda environment.

We have two long-term goals. The first one is an implementation of a Semantic Web-specialized distributed parallel data-storage, which can significantly improve the behavior and performance of the Semantic Web repository.

As the second long-term goal, we plan to interconnect diverse semantic repositories, possibly with different implementation. Such interface-based loosely coupled network could become a nucleus of really usable semantic web, both for academic and practical purposes.

Acknowledgement

This work was supported by the Grant Agency of the Czech Republic, grant number 201/09/0990 - XML Data Processing.

References

[1] D. Beckett. Turtle - terse rdf triple language, 2004. http://www.dajobe.org/2004/01/turtle/.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[3] J. Broekstra and A. Kampman. SeRQL: A Second Generation RDF Query Language. In *Proceedings of the Workshop on Semantic Web Storage and Retrieval*, Netherlands, 2003.

[4] J. Broekstra, A. Kampman, and F. Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the First International Semantic Web Conference*, pages 54–68, Italy, 2002.

[5] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[6] J. Dokulil. Evaluation of SPARQL queries using relational databases. In I. Cruz, editor, *5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006*, volume 4273 of *Lecture Notes In Computer Science*, pages 972–973, 2006.

[7] J. Dokulil, J. Tykal, J. Yaghob, and F. Zavoral. Semantic web repository and interfaces. In *SEMAPRO07: International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 223–228, Los Alamitos, 2007. IEEE Computer Society.

[8] L. Galambos. Robot UDP Transfer Protocol, 2007. http://www.egothor.org/RFC/RUTP-v02.pdf.

[9] L. Galamboš. Dynamic inverted index maintenance. *International Journal of Computer Science*, 2006.

[10] J. Galgonek. Query languages for the semantic web. Master thesis at Charles University in Prague, 2008.

[11] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. Xr-tree: Indexing xml data for efficient structural joins. In *Proceedings of the 19th International Conference on Data Engineering (ICDE03)*. IEEE, 2003. 1063-6382/03.

[12] M. Olson and U. Ogbuji. Versa, 2002.

[13] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Working Draft, 2005. http://www.w3.org/TR/2006/WD-rdf-sparql-query-20060220/.

[14] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Applied Ontology*, pages 77–106, 2005.

[15] T. Vitvar, A. Mocan, M. Kerrigan, M. Zaremba, M. Zaremba, M. Moran, E. Cimpian, T. Haselwanter, and D. Fensel. Semantically-enabled service oriented architecture: Concepts, technology and application. *Journal of Service Oriented Computing and Applications*, 2007.

[16] S. Wang, Y. Guo, A. Qasem, and J. Heflin. Rapid benchmarking for semantic web knowledge base systems. Technical Report LU-CSE-05-026, CSE Department, Lehigh University, 2005.

[17] J. Yaghob and F. Zavoral. Semantic web infrastructure using datapile. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence and Itelligent Agent Technology*, pages 630–633, Los Alamitos, California, 2006. IEEE. ISBN 0-7695-2749-3.

[18] N. Zhang, V. Kacholia, and M. Tamer. A succinct physical storage scheme for efficient evaluation of path queries in xml. In *Proceedings of the 20th International Conference on Data Engineering (ICDE04)*. IEEE, 2004. 1063-6382/04.

[19] W3C Semantic Web Activity Statement, 2001. http://www.w3.org/2001/sw/Activity.

[20] The Apache Software Foundation. http://www.apache.org.

[21] Microsoft Internet Information Services. http://www.microsoft.com/WindowsServer2003/iis.

[22] Jena A Semantic Web Framework for Java. http://jena.sourceforge.net/.