IT 10 065
Examensarbete 30 hp
November 2010

Wrapping SparQL Query Services

Mikael Lax

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Wrapping SparQL Query Services

Mikael Lax

In a world where information is spread out over innumerable data sources it is often desirable to be able to access data in a unified manner regardless of how it is stored. Amos II is an object-oriented mediator database that can fulfill this role by defining wrappers for foreign data sources and allowing the foreign data to be queried through the query language AmosQL. In this project a facility in Amos II known as the translator API is studied and utilized in order to create a wrapper for the specialized RDF query language SparQL. A major problem encountered when wrapping a foreign data source is the low performance incurred when a query is executed entirely by the mediator, unaided by the foreign data source. The translator API approaches this problem by allowing all or part of an AmosQL query to be executed on the foreign data source by translating the query into the language used by the query engine of the foreign data source. We demonstrate that by using the translator API, it is possible to use only AmosQL and still achieve highly efficient queries to a foreign SparQL enabled data source.

Handledare: Silvia Stefanova
Ämnesgranskare: Tore Risch
Examinator: Anders Jansson
IT 10 065
Tryckt av: Reprocentralen ITC

Table of Contents

Introduction

The Semantic Web is the name given by the W3Cs organization to add meaning in ways computers can understand to the vastness of documents on the Internet (1). A problem for computers is that documents on the web are generally unstructured and meant only for humans. The idea is to annotate online articles and resources with metadata to allow for computers to search, combine and present information in unprecedented ways. To accomplish this, the W3C has put forth a large array of technologies and standards. The goals of the Semantic Web initiative are lofty and far-reaching and we will only focus on two of the key technologies involved. First up is the RDF metadata model; initially published as a recommendation by the W3C in 1999 (2), RDF offers a deceptively simple format for declaring statements about web resources. The other Semantic Web technology studied is SparQL (3) which is a specialized query language for data stored as RDF. SparQL became an official recommendation by the W3C in early 2008 and is lauded as a key Semantic Web technology. Though many query languages for RDF have been made, it's the hope of the W3C that SparQL will become the de facto standard for now and the future. RDF data is often stored in relational databases which has caused some attention to be paid towards translating SparQL to SQL to achieve efficient queries (4). In addition, special purpose databases geared towards RDF with SparQL query capabilities are in existence allowing native execution of SparQL queries (5).

Databases have become a commonplace technology in the modern world. It is difficult to find any walk of life, whether in industry or academia, that doesn't store information in databases of some sort. A problem that can occur when data is spread across such a prodigious smattering of databases is when we need to combine data from two or more sources that may not be immediately compatible with each other. One way to approach this problem is with a *mediator* database that can query a multitude of disparate sources through a unified query language and combine and present the results. One such system goes by the name Amos II and is developed at the database laboratory at Uppsala University (6). Amos II is an object-oriented, mediator database with a functional data model as well as its own query language called AmosQL. Amos II lets the user write *wrappers* to incorporate external data sources into Amos II; a large number of wrappers for a multitude of varied sources have already been written (7). One issue that comes up when wrapping a foreign data source is how the division of labor between the foreign data source and Amos II is supposed to be carried out. A wrapper written previously for relational databases examined this issue by developing a *translator API* for performing query translation (8). The query translator is based on the quite reasonable idea that the foreign data source is well optimized for querying its own data. Through the use of the translator API a query in AmosQL can be translated into the language used by a foreign data source; this lets Amos II take care of combining and presenting the results of a query while still off-loading a large part of the computational burden of performing the query on the foreign data source.

In this project a wrapper, complete with a query translator, is presented that allows for queries written in AmosQL to be automatically translated into SparQL and sent to a SparQL enabled service. The project will also hopefully be useful as a guide for people looking to use the translator API to wrap other data sources as well.

Background

This section will offer introductory explanations of the different technologies and software used in the project. We will begin with the semantic web technologies and then leap into the Amos II system.

RDF

RDF, or the Resource Description Framework, is a data model for declaring metadata about resources (9) (10). RDF is based on the idea of establishing facts about resources in the form of RDF triples. An RDF triple consists of a *subject*, a *predicate*, and an *object*. The subject is a globally unique identifier for a resource, the predicate defines a property and the object is the value of this property. While often described as a set of triples, an RDF dataset logically represents a directed graph and is commonly referred to as an *RDF graph*. In such a graph the subjects and objects are the nodes and the predicates correspond to the arcs.

RDF uses Uniform Resource Identifiers (or URIs for short) as identifiers. A URI, though much can be said about it, is simply a sequence of characters commonly used to identify a resource over the Internet (11). RDF can thus be used to describe metadata about web resources. While RDF does require that URIs be used for identifiers, it does not require that they actually resolve to anything on the Internet, so RDF is not limited to simply describing web resources. Any part of the RDF triple can be a URI and in particular the predicate is required to be one.

To designate the object (the value part of the triple) RDF also has *literals*. A literal is a string and can be either *plain* or *typed*. A plain literal, though lacking a type, can have an optional *language tag* which has implications when determining if literals match each other. A typed literal has a *datatype* URI attached to it that constrains the literal by specifying that it must represent a value in the lexical space defined by the datatype. This, somewhat obtuse, way of introducing datatypes is brought about by the fact that RDF itself does not define any datatypes (except one type, called XMLLiteral, which is not covered here). Instead, RDF has an abstraction of the datatype concept which allows users to define their own datatypes. Nevertheless, it would be unreasonable to expect users to define their own datatypes for extremely common primitives. The W3C thus defers to the document 'XML Schema Part 2: Datatypes' (12) that defines all the usual suspects such as the integers and doubles we all know and hold dear.

Examples of literals:

"Hello"	Plain, untyped literal
"Hello"@en	Plain, untyped literal with a language tag
"42"	Plain, untyped literal
"42"^^"<http://www.w3.org/2001/XMLSchema#integer>"	A literal typed as an integer
"1.1"^^"<http://www.w3.org/2001/XMLSchema#double>"	A literal typed as a double

The last entities we need to know about are called *blank nodes*. A blank node is used to mark a resource that is currently unidentified. A resource can, for example, be unidentified because it's not meaningful for it to be named or simply because a value for it is not yet known. A common usage of

blank nodes is to store a composite value in a structured way. For example, a person's address might be represented as a string but this can be unstructured and lead to ambiguities. In RDF this is solved by considering the address as a resource of its own and then making statements such as street name, zip code and so on about this new resource. Since it might not be useful to give this artificially introduced resource a universal identifier, it is a prime candidate to be a blank node.

At its core, RDF is specified by an abstract syntax and semantics. When dealing with the real world an actual serialization format needs to be used. One such format laid down by the W3C is called RDF/XML (13) and specifies an XML syntax for representing RDF data. The XML format is convenient for programs to work with; for humans, there's the Turtle format (14) which is easier to write by hand. The simplest format is probably N-Triples (15) which pretty much stores RDF as a long series of data rows.

SparQL

While storing data as RDF is all well and good, once we have that, we would also like to be able to search the RDF graph in efficient, concise, and intuitive ways. Enter SparQL, a query language designed for RDF graphs (3). Writing queries in SparQL means specifying a subgraph structure to match against, as well as specifying what information should be extracted from the resulting matches. That SparQL has been designed from the start with RDF in mind is clearly visible from its syntax which deals directly with RDF triples. Since we will be generating queries in SparQL a short introduction to the relevant syntax and semantics will be provided.

Syntax and semantics

For the examples we'll assume a triple store that uses the FOAF format (16) to store data about people. FOAF is an example of RDF put to use to establish a vocabulary to make statements about persons. A driving idea behind FOAF is to describe relations between people in a machine-readable way, hence the name Friend of a Friend. In this project, however, we only use FOAF to spruce up the examples a little. Let's start with a simple query:

```
select ?name ?age
where { ?s <http://xmlns.com/foaf/0.1/name> ?name .
        ?s <http://xmlns.com/foaf/0.1/age>  ?age }
```

This would return the name and age of all the subjects. Note that when we say "all", we mean all subjects that use the FOAF vocabulary. We make this distinction because there is no requirement that a triple store represents data in any structured way at all. The part following the *where* keyword is called a *basic graph pattern* and it consists of a set of *triple patterns*; it is the subgraph structure used to check for matches. Query variables begin with a '?' (or a '$') and are wildcards that can appear anywhere in a triple pattern. A query variable has global scope across a query, meaning that if a variable occurs more than once, it must refer to the same item. This is why in the query above, the result is the name and age of all subjects and not all the combinations of all names and ages. Query variables when situated after the select keyword specify which variables in the basic graph pattern that constitutes a solution.

We can constrain the solution set by using the *filter* keyword:

```
select ?name ?age
```

```
where { ?s <http://xmlns.com/foaf/0.1/name> ?name .
        ?s <http://xmlns.com/foaf/0.1/age>  ?age .
        filter(?age > 18) }
```

This would return the name and age of all subjects whose age is greater than eighteen. Note the use of the datum 18. Previously we said that a literal was a string, possibly typed, but the above doesn't look like a string (i.e. no quotation marks), typed or not. The resolution to this is that SparQL is designed with the datatypes put forth in the document 'XML Schema Part 2: Datatypes' (12) closely in mind. Operators in SparQL are defined over these datatypes and the language provides a syntactic shorthand for them. The 18 above is equivalent to the typed literal "18"^^<http://www.w3.org/2001/XMLSchema#integer>. Similar shortcuts come built-in for other primitives, like booleans and decimals, as well.

We can filter using regular expressions as well:

```
select ?name ?age
where { ?s <http://xmlns.com/foaf/0.1/name> ?name .
        ?s <http://xmlns.com/foaf/0.1/age>  ?age .
        filter(?age > 18 && regex(?name,"^A")) }
```

This would return the name and age of all subjects whose age is greater than eighteen and whose names begin with 'A'. Filtering with the *regex* function works only on literals without language tags; to filter on a language tagged literal the *str* function must be used to treat the literal as a plain literal (*str* can be used to match on URIs as well). For the interested, the regular expression language is defined in the W3C recommendation 'XQuery 1.0 and XPath 2.0 Functions and Operators' (17). For the uninterested, the regular expression language is fairly similar to the one laid down by the Perl programming language.

At this point, we have covered all the aspects of SparQL syntax and semantics needed for our purposes in this project, but we can still mention some other parts of the language that are commonly seen.

SparQL requires all URIs to be absolute (i.e. fully specified) which can result in a lot of typing when writing queries by hand. To abbreviate query writing we can use *prefix* and *base* URIs. Any number of prefixes can be used but only one base. The following three queries are all equivalent to each other:

```
select ?name ?age
where { ?s <http://xmlns.com/foaf/0.1/name> ?name .
        ?s <http://xmlns.com/foaf/0.1/age>  ?age }


prefix foaf: <http://xmlns.com/foaf/0.1/>
select ?name ?age
where { ?s foaf:name ?name .
```

```
        ?s foaf:age  ?age }
```

```
base <http://xmlns.com/foaf/0.1/>
select ?name ?age
where { ?s <name> ?name .
        ?s <age>  ?age }
```

In the scary outside world, the data we encounter may be incomplete; SparQL can deal with this by allowing all or some of the solutions to be declared as *optional*. The following query retrieves all names and, if available, e-mail addresses:

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
select ?name ?email
where { ?s foaf:name ?name .
        optional { ?s foaf:mbox ?email } }
```

Without the use of the keyword *optional* above, we would get no results for people who lack an e-mail address. Optional graph patterns can be filtered as well:

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
select ?name ?email
where { ?s foaf:name ?name .
        optional { ?s foaf:mbox ?email .
                   filter(regex(str(?email),"@hello.com$","i")) } }
```

If we don't want an actual solution set but just want to know if a solution exists at all, we can use an *ask* query. The following query asks if anyone named Mikael exists in the dataset. The answer to an ask query is always either true or false.

```
ask { ?x <http://xmlns.com/foaf/0.1/name> "Mikael" }
```

There are a lot more features in SparQL than this, but with an eye towards brevity we will stop here. Next we'll briefly mention how communication with a *SparQL service* is handled; how are the queries sent and results retrieved. SparQL services are commonly presented as web services with queries and results being transmitted using the HTTP protocol. The specifics of how to interact with a SparQL service is laid down by the document SparQL Protocol for RDF (18) but it's primarily of interest to developers of SparQL services (i.e. not us). We'll be using software to take care of the lower level details; all we need to be able to do is encode a query in the form of a URL and parse the result. The result of a SparQL query is dictated by the descriptively titled document SparQL Query Results XML Format (19).

XML result format

The result format can be divided into two parts (19). The *header* that introduces the query variables used and the *results* section that states sets of bindings for the variables. In the header the variables will be encountered in the order they appear in after the select statement in a SparQL query.

Example header:

```
<head>
  <variable name="name"/>
  <variable name="age"/>
  <variable name="mbox"/>
</head>
```

The next element is the *results* element. Any number of *result* elements may appear as child elements of results (carefully noting the distinction between the singular and plural). In turn, each result element has, in no particular order, a number of bindings as its own children.

Example results:

```
<results>
  <result>
    <binding name="name">…</binding>
    <binding name="age">…</binding>
    <binding name="mbox">…</binding>
  </result>
  <result>
    <binding name="age">…</binding>
    <binding name="name">…</binding>
  </result>
</results>
```

In the above results we can see that variable *mbox* must have been declared as optional by the SparQL query since it doesn't appear in the second result; this is how optional results are seen in the results format. Next we will look at how the bindings are specified. RDF gives us three distinct types of result values for the bindings: URIs, literals and blank nodes. The literals can be further divided into three classes: plain literals, literals with language tags and typed literals.

Example bindings:

```
<binding name="name">
  <literal>Mikael</literal>
```

```
</binding>
<binding name="age">
  <literal datatype="http://www.w3.org/2001/XMLSchema#integer>
    27
  </literal>
</binding>
<binding name="mbox">
  <uri>mikael@the.internet.com</uri>
</binding>
<binding name="text">
  <literal xml:lang="es">¿Cómo estás?</literal>
</binding>
```

Finally we should mention that the above format is completely eschewed by the boolean valued ASK queries. An ASK query has only a true or false answer and does not declare any variables in the header. Furthermore, it has a boolean element instead of the results element. An answer to an ASK query, and here we also show the complete document structure, can look like the following example:

```
<?xml version="1.0?">
  <sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head/>
  <boolean>true</boolean>
</sparql>
```

Jena and Joseki

Jena is an open source framework in Java for developing semantic web applications (20). Jena hails from a (now discontinued) research programme into the semantic web undertaken by Hewlett-Packard. Among other things it provides an API for manipulating RDF graphs, serialization of RDF data to and from the RDF/XML, Turtle and N3 formats as well as a SparQL query engine. Jena can also achieve persistent storage by storing the RDF triples in a back-end relational database.

We won't be using Jena directly in this project. Joseki, which is employing Jena and was also originated by Hewlett-Packard will be utilized instead (21). Joseki is a triple store and a web server that supports the HTTP bindings of the SparQL protocol. It allows SparQL queries to be sent as HTTP requests and the results to be returned in the SparQL Query Result XML format recommended by the W3C. Joseki is a good fit for this project since it allows us to perform queries on arbitrary datasets loaded from local files, which is convenient when developing and testing the wrapper.

Amos II

This section will give a crash course on the data model and query language of Amos II and how to use it. There is certainly more to the system than what is shown here. The data model in Amos II is based on *objects*, *types* and *functions* (22) (23).

All data stored in Amos II will be an object of some sort. We have two distinct classes of objects: *literals* and *surrogates*. Literals are built-in, fundamental objects maintained by the system such as integers and charstrings. Any object that isn't a literal is called a surrogate. Unlike a literal, a surrogate isn't self-descriptive but is represented by an object identifier called an *OID*. All user-defined objects are surrogates.

Types are used to classify objects into different sets that determine the semantics an object may exhibit. All objects in Amos II belong to one or more types, but always at least one, meaning that all objects are typed. Types may inherit from other types meaning, in regular object-oriented fashion, that an instance of a subtype may act as an instance of any of its supertypes. Multiple inheritance is supported as well, so a subtype isn't limited to just one direct supertype.

Examples of declaring types:

```
create type Person;

create type Employee under Person;

create type Manager under Employee;
```

Functions implement the actual semantics of our objects and establish relations between objects. Functions can be identified by their *signature* which is the combination of its name, argument types and result types. We have three different classes of functions: *stored*, *derived* and *foreign* functions.

Stored functions record mappings between objects and are what we use to add attributes to our types. Stored functions get their name because their results are all stored physically in the database.

Examples of stored functions:

```
create name(Person) -> charstring as stored;

create salary(Employee) -> integer as stored;

create bonus(Manager) -> integer as stored;

create employees(Manager) -> bag of Employee as stored;
```

Thanks to inheritance not just persons, but anything inheriting from person (type *employee* and *manager*), will have names as well. The same logic extends to the salary attribute but only the manager gets a bonus and employees to command since that's the most specific type available for those attributes. The employees of a manager are declared as being a *bag*; this simply means there can be more than one of them. Bags are unordered and allow duplicates. If we need ordered collections there is a collection type named *Vector* for that purpose.

Once we have declared some stored functions for a type we can begin creating instances of the type. The set of all instances of a type is called the type's *extent*.

Examples of creating instances:

```
create Employee (name, salary)
  instances ("Bob",100), ("Kurt",125), ("Mia",100);
create Manager (name, salary, bonus)
  instances :alice ("Alice",150,9999);
```

An identifier like *:alice* above is called an *environment variable*; we can use them as convenient shortcuts to access the instance like in the following line that puts our little workforce under Alice's command (making sure not to add Alice herself):

```
add employees(:alice) = (select e from Employee e
                         where e != :alice);
```

Derived functions are basically AmosQL queries stored as functions. These functions should be free from side effects and, like ad-hoc queries, begin with a select statement. Let's take a moment to look at AmosQL. The anatomy of an AmosQL query is as follows:

```
select [results]
from [types]
where [conditions];
```

The types specify the extents the query should operate on; if more than one type is present the Cartesian product of the extents is considered. The conditions serve to constrain the size of the result set, with no conditions the entire extent (or its Cartesian product) is used to build the result set. The results part of the query specify what aspect of the resulting objects we are interested in; this usually involves the attributes of the objects in some way and we can also do things like arithmetic and other function calls to form the final result.

Examples of AmosQL queries:

```
select name(m), name(e)
from Manager m, Employee e
where e in employees(m);
```

The above query gives us the names of all managers and their employees. Here a result consists of two values; when this is the case each result is returned as a *tuple*. Putting it simply, a tuple is an ordered set with a fixed size and fixed element types.

```
create function net_worth(Employee e) -> integer as
       select salary(e);
create function net_worth(Manager m) -> integer as
```

```
        select salary(m) + bonus(m);
```

These two functions show the use of function overloading; even though they have the same name, no ambiguity exists since the type of the input arguments are used to further distinguish the function. If we query for the net worth of all employees in this case, we would see that the correct function would be chosen for the employees that are also managers (an example of late binding). Furthermore it can be noted that when an object is introduced as part of a functions signature, it can be elided from the actual query.

Foreign functions are functions implemented in a language other than AmosQL. Options for foreign languages provided by Amos II are C, Java and ALisp (a Common Lisp inspired lisp dialect used extensively inside Amos II). Foreign functions are commonly used to interface with external data sources but they're not limited to that; we can use them to do anything we feel would be cumbersome to do in AmosQL. A foreign function is allowed to have side effects, but if it's to be used in queries it's expected to be side effect free.

```
create function frob(integer) -> charstring as foreign "frob";
```

Stored and derived functions have the useful property of being invertible. Consider the following snippet that selects the salary of the manager named "Alice":

```
select salary(m) from Manager m where name(m)="Alice";
```

The function name is declared as having a *charstring* result value, but here, the result is already provided and what's being sought is the manager *m*. We say that such a function is *invertible*. Foreign functions can be made invertible as well by declaring them as *multi-directional*:

```
create function frob(integer) -> charstring as multidirectional
        ("bf" foreign "frobbf" cost {1,1})
        ("fb" foreign "frobfb" cost {10,1});
```

The strings "bf" and "fb" are called *binding patterns*. They specify which of the variables passed to the function that must be free (f) or bound (b) for that implementation of the function to be selected. A free variable can be seen as an output variable that the function should produce a value for. A bound variable can be seen as an input variable that, in one way or another, has a value already provided for it. We also have the concept of *coverage* in a binding pattern; we say that a free variable covers a bound variable, meaning that any variable marked as free may also take a bound variable. This is possible because the system can always call the more general implementation and verify the result value. If we make a call like 'frob(1)="A"' with the example function above the system will call "frobbf" (because it has lower cost) and check if the result matches "A". If we have an implementation for the "ff" pattern, which would produce a complete mapping of all input and output values, we would automatically gain the functionality of all the conceivable binding patterns. Such an implementation could have a potentially huge result set, so even if we have it, we could still want to provide specialized implementations for the more specific binding patterns for performance reasons.

The *cost* is optional to specify as a hint to aid the query optimizer. The cost is designated in execution cost and fanout (result set) size. The example function above is expected to have a single result value in both directions and performing the operation in reverse is roughly ten times as expensive relative to the forward direction. It's also possible to give the cost as a function if we want to compute the cost dynamically at query compilation time.

Making a SparQL wrapper

A wrapper can be logically divided into two separate components: First, there's interface to the foreign data source responsible for the low level communication between Amos II and the data source. Second, there's the translator which rewrites queries from AmosQL to the language of the data source. While translating queries from one language to another might sound complicated at first, the translator API simplifies things for us. In particular, one does not need to have any understanding of how AmosQL syntax is parsed and only a shallow understanding of how it is compiled. We will begin our excursion into wrapper land by looking at a simple usage example of the finished wrapper. Then we transition into showing how, with a minimum of work, we can achieve transparent but untranslated access to a foreign data source using some basic tools in Amos II. Then, at last, we segue into the meat of the project where we present the translation API and demonstrate how it can be used to accomplish high performance queries by pushing large amounts of the necessary work to be executed onto the foreign data source instead of doing it in Amos II.

A simple usage example

It's difficult to perform queries over non-existent data, so let's first define a small toy database to use as an example throughout this chapter.

```
<http://person/Mikael> <http://property/firstname> Mikael

<http://person/Mikael> <http://property/lastname> Lax

<http://person/Mikael> <http://property/age> 27

<http://person/Bosse> <http://property/firstname> Bosse

<http://person/Bosse> <http://property/lastname> Andersson

<http://person/Bosse> <http://property/age> 42
```

The first step is to instantiate a data source and connect it with a SparQL endpoint. This is done as follows:

```
set :people = sparql("people","http://url.to/people");
```

Now we can refer to people as if it was an ordinary AMOS II type named "people" using the functions *s*, *p*, and *o* (for the subject, predicate, and object); these are the functions we would expect to have defined for an RDF triple and they are created for us automatically when we instantiate the data source. The environment variable *:people* contains the OID of the datatype *people* itself, there is rarely any need for the user to carry this around since it's not needed for doing queries.

To perform a query we can now use AmosQL directly. A query to request the first and last name of our highly fictitious people would look like this:

```
select o(p_a), o(p_b) from people p_a, people p_b
where p(p_a) = '<http://property/firstname>' and
      p(p_b) = '<http://property/lastname>' and
      s(p_a) = s(p_b);
```

This gives us as a result the following bag of tuples:

```
<"Mikael","Lax">
<"Bosse","Andersson">
```

As an added bonus, we can now get SQL access to our people database as well. AMOS II has a built-in way of enabling SQL access to a function by using a magic naming convention (a "#" sign in front) when defining it.

```
create function #magic() -> <charstring s,
                             charstring p,
                             charstring o>
as select s(ppl), p(ppl), o(ppl) from people ppl;
```

We can now use SQL to access the people as follows:

```
sql("select p_a.o, p_b.o from magic as p_a, magic as p_b
     where p_a.p = '<http://property/firstname>' and
           p_b.p = '<http://property/lastname>' and
           p_a.s = p_b.s");
```

This gives us as a result the following bag of vectors:

```
{"Mikael","Lax"}
{"Bosse","Andersson"}
```

Dealing with foreign objects

In this section we will look at the bare essentials needed to be able to query a foreign data source using AmosQL (23). The queries will not be translated, which means that all filtering and join operations will be performed by Amos II rather than by the back-end data source query engine, but the tools used here are relevant for doing translated queries as well.

Proxy objects

Amos II is an object-oriented database and thus stores data in the form of objects with attributes. By using wrappers this can be extended to foreign objects in foreign data sources in the sense that the foreign objects can be queried using AmosQL as if they were regular objects, but not in the sense that these objects are actually stored in Amos II. A foreign object when constructed in Amos II is called a *proxy object* since it merely serves as a handle to the real data and any request for the values of the proxy object's attributes will result in accesses to the foreign data source. Note that a foreign data source may not even store its data as objects in any traditional sense, but it must still be possible to view the foreign data as collections of some form of objects with attributes if we want to wrap the data source using Amos II.

A proxy object is defined by a *mapped type*. When declaring a mapped type we always need to specify four things: a name for the type, a list with names and types for the attributes of the mapped type, a list with a subset of those attributes that make up the key of the type (i.e. the attributes necessary to uniquely identify an object of this type) and finally a function that returns all the attributes of all objects of this type as a bag of tuples from the foreign data source. This function is called the *core cluster* and will be explained soon. The name for the mapped type is how we will access it in Amos II queries, just as with regular objects defined by regular types. From the list of attributes the system will automatically construct a number of functions that select the different components of the tuples returned by the core cluster. So to be clear, the first attribute will therefore be the first element in the tuple returned by the core cluster, the second attribute will be the second element, and so on. It's up to the user to both define the core cluster and also to ensure that the key is unique. The key needs to be unique in order to establish object identity which is important for mapping the results of the core cluster to proxy objects.

The Core Cluster

The core cluster, as previously mentioned, is a function that returns all the attributes of a specific mapped type from a foreign data source as a bag of tuples. Each attribute can be of any type so the general signature of the core cluster is as follows:

```
foo_cc() -> bag of <t0 v0, t1 v1, ..., tn vn>
```

By convention, the core cluster tends to have its name suffixed with "_cc". Since the core cluster will be communicating with a foreign data source, it's customary for it to be written (wholly or partially) in a language suited for this purpose such as C or Java.

To obviate the use of the core cluster consider the following snippet:

```
select name(p) from person p
where age(p) > 30;
```

If *person* is a regular type in Amos II this would give us the names of all persons older than 30 stored in the database. The same should also be true if type *person* was defined by a mapped type, in which case *p* would be a proxy object, and we could think of the above snippet as a thin veneer for something amounting to the following:

```
select name
```

```
from charstring name, integer age
where <name,age> in the_core_cluster() and
      age > 30;
```

The example is meant only as illustrative but the basic idea remains: the core cluster is the fundamental unit of communication with the foreign data source and an introduction of a proxy object in a query will, below the surface, be rewritten by Amos II to call the core cluster specified by its mapped type.

It's pertinent to keep in mind that a mapped type defines a proxy object that, through the core cluster, is connected to a specific data source. This means that we don't create just one mapped type to describe a proxy object in all data sources; each data source will need its own mapped type (and, by extension, its own core cluster) even for the same type of proxy object. In practice, this means that each mapped type must get its own unique name and we should also ensure that all core clusters return the proxy objects attributes in the same order for all proxy objects considered to be of the same type.

A SparQL example

A data source can be accessed through as many mapped types as we can imagine discrete object mappings for, but in a SparQL wrapper we only have one type of foreign object: the RDF triple. The components of the triple will be stored as charstrings and the key will consist of all three components since no strict subset of an RDF triple can be guaranteed to be unique.

In this project a call to a core cluster will be a request for the subject, predicate and object of all RDF triples in the triple store. To implement the core cluster we need a function that can send a query as an HTTP request and parse an XML document containing the result; for this project Java was chosen for this purpose. We don't write the entire core cluster in Java, instead we write a general query facility in Java that will be useful later on as well. Its signature is as follows:

```
sparql_query(charstring query, charstring address) -> bag of vector
```

In order to perform a query, the function first encodes the query as a URL and then sends it to the address. In Java this is a one line operation. Encoding the query basically just means that characters that shouldn't appear in a URL like the curly braces in a SparQL where-clause are replaced or percent-escaped. For example, spaces are replaced with plus signs and the {-character becomes %7B. The other task the function does is parsing the result XML format that was described earlier.

With this we can create a core cluster function as following:

```
create function stuff_cc() -> <charstring s,
                               charstring p,
                               charstring o>
as select s, p, o
where {s,p,o} = sparql_query("select ?s ?p ?o where {?s ?p ?o}",
                             "http://url.to/stuff");
```

With core cluster in hand we can now create a mapped type as follows:

```
create_mapped_type("stuff",{'s','p','o'},{'s','p','o'},"stuff_cc");
```

Note that we don't have to specify the types for the attributes since the system can infer all the types from the signature for the core cluster. With this Amos II will automatically set up all the machinery we need to send AmosQL queries to our mythical "stuff" triple store. Example:

```
select o(s1), o(s2) from stuff s1, stuff s2
where p(s1) = "some predicate" and
      p(s2) = "another predicate" and
      s(s1) = s(s2);
```

The Translator API

While the previous example does give us AmosQL access to a foreign data source, it has one fatal flaw: performance. The self-join operation is especially expensive to perform on the Amos II side, as will be explored in the later section on performance. For a SparQL wrapper this is a serious issue since SparQL queries in general have large numbers of implicit self-joins in them. In order to achieve acceptable performance on larger datasets it's going to be necessary to translate the AmosQL queries to SparQL and have the SparQL endpoint (Joseki, in our case) execute them instead. A major motivation for query translation is therefore not just the ability to do transparent queries, but the ability to do transparent queries with reasonable performance. This section is divided into two primary parts; first an explanation of the concepts used by the translator API (8), and then, an extensive example of how to write a translator in practice. This example is, of course, going to be our much touted AmosQL-to-SparQL translator.

Data sources

Not being content with only mapped types we will also introduce the data source as a distinct entity. In Amos II a data source is declared by inheriting from the type *Datasource* and it serves as a place to centralize any information that is general to any instance of the data source we are targeting. The data source also has the practical role of having the translator API built around itself, so in a sense, we could say that a data source will know how to translate the queries that are aimed at it.

When using the data source to store information used by the translator it's common to use what's called the *property list* of the instance. Anything with an OID can have a property list, which is a list of key-value pairs used by the system to store metadata on things like types and functions. When we refer to storing or recording things "on" a data source or function, we mean adding the data to the property list.

So far we have been cavalier about what translating a query actually entails. Natural questions arising are: What parts of a query are translated? How are they translated? What does it mean to successfully translate a query? We will now delve into the details of these questions.

Capabilities

To help answer "What parts of a query are translated?" we introduce the concept of a *capability*. Informally, a capability is some operation that Amos II can perform that we also can find a counterpart for in the foreign data source. Examples would be tasks such as doing arithmetic comparisons and other forms of filtering operations.

More formally, a capability is specified by the following: a data source, a function, a binding pattern for the variables of the function, and an *absorbent*. The need to provide the data source should be clear since it makes no sense to think of capabilities separate from data sources. If a data source doesn't support a certain operation it would be futile to try to perform that operation on the data source. The function represents the operation on the Amos II side and the corresponding binding pattern tells under what combination of variable bindings it's a candidate for translation. The absorbent will be explained in just a little while.

A capability can be defined either on a data source's type, making it general to all instances of that data source, or it can be defined on an instance of a data source making it specific to only that instance. The latter case might seem puzzling; assuming all data sources of the same type are all equally capable, why would a capability still need to be specific to just a single instance of the data source? To see this, consider the core cluster. The core cluster is a function, and as such, can be seen as a capability that establishes or verifies the bindings of the variables in a query. Since each data source will be accessed through its own mapped types, and each mapped type has a unique core cluster function, it follows that each core cluster must get its own capability. As we will find out, translating calls to the core cluster is going to be of prime importance in a translator.

At this point, we can deduce that translating a query is a step-by-step process and the size of each step is a function call. The idea is that as many of these steps as possible should be combined to eventually form a single query meant to be sent to the data source. This also carries the tautological implication that it's not necessarily an entire AmosQL query that is translated, but only the parts that we are able to translate with the remaining parts still being done on the Amos II end. Our ability to translate a part of a query (i.e. a function) hinges thus on i) whether the data source has that capability, and ii) even if the data source does have that capability, whether any of its operands originate in the data source. The job of finding these eligible functions is handled for us by the translator API, but the burden of translating these functions falls upon our shoulders. To help us with this we have absorbents.

Absorbents

An absorbent is a function that knows how to translate a specific function associated with a capability. While an AmosQL query is being compiled, absorbents are called automatically by the translator API on the functions in the query that, through the use of capabilities, have been deemed translatable. The implementation of an absorbent is left entirely as the province of the programmer but all absorbents will be passed four arguments. The first argument is the data source that belongs to the capability that invoked this absorbent. The second argument contains the function being translated (its OID) and its variable list. The variable list contains variable identifiers for the input and output arguments to the function. The last two arguments are called the *environment* and the *accumulator*; these are fairly simple constructs whose explanation will be incorporated in the upcoming section.

The work an absorbent needs to do depends a lot on the nature of the foreign data source we are targeting but we can still provide a few guidelines as to how an absorbent is usually expected to work. An absorbent in action tends to be directed predominantly by what it knows about the variables used by the function. Recall that translation takes place at the time of compilation, so luxuries such as values for our variables will be withheld from our grasp (the only exception being when those values are provided as constants). Nevertheless, we can still know a presentable amount about these variables; namely: whether the variable is bound or free, the data source associated with its binding, its type, its origin (the specific function that bound it), and its *entity*. A use for the origin was not found in this project, but the provision to record it is there. The entity is a property that explicates the meaning of a variable as understood by the foreign data source. As an example, the entity in the context of this project would therefore be used to record if a variable was the subject, predicate, or object component of an RDF triple. All this information is begging to be stored somewhere, and that is where the environment enters the picture. The environment is a lookup table which is indexed by variables; it gets passed along between absorbents as translation is underway. The environment is not a read-only object from our perspective; when absorbing calls to the core cluster it's necessary to set up bindings for the free variables that are sought, which requires us to record their binding status and data source. It's important that the binding and data source are recorded properly since this information is used to direct the translation process. The entity and origin, on the other hand, are currently only provided as an aid to the programmer and we don't have to record them unless we know we need them. We use *absorbing a function* as synonymous with translating it. Absorb into what? This brings us to the final argument passed to an absorbent: the *accumulator*. The accumulator is an empty box handed to the programmer for personal use. As with the environment, the accumulator will be passed down the line of absorbents with the expectation that the functions encountered will accrete in it. Due to the large and varied amount of data sources we might run up against, there are no formal requirements placed on the accumulator; it's simply up to the programmer to figure out, at each step, what needs to be put into the box in order to later create a query for the foreign data source from it.

When a query is being translated, the query optimizer will try many different orderings of the absorbents. This means that when writing absorbents we should try not to rely on the absorbents being called in a specific order. As a rule of thumb, an absorbent should not maintain any global state, but instead store everything it needs in the environment and the accumulator.

Since there are so many different data sources and capabilities out there, the translator API recognizes that an absorbent may not always be able to do absorb a function even if it's deemed translatable through its capability specification. If an absorbent, for any reason, discovers that it can't absorb a function it can signal failure and let the operation be performed by Amos II instead.

The default absorbent

One absorbent that deserves special mention is the absorbent for the core cluster. This absorbent is so ubiquitous that it's commonly called the *default absorbent*. In order for translated access to a mapped type to be possible a default absorbent must be provided for its core cluster. The default absorbent is the only absorbent that doesn't have to be part of a capability, although we may often want it to be. The default absorbent must always be stored on the core cluster itself (i.e. its property list), this stems from the purely practical matter that the default absorbent serves as the entry point to the translation process; since it's the core cluster that delivers our foreign data it makes sense for

translation to start there. This means that the default absorbent can be called in two ways: as the starting point for a translation or as part of a capability. Viewed by the author of a translator this distinction is largely irrelevant since the absorbent itself doesn't know how it was invoked. If the core cluster is not made to be a capability it means that every appearance of a core cluster call in a query will result in a separate translation process being started. This means that joins can't be translated and must be performed by Amos II instead, so the only time when we realistically would not want the core cluster to be a capability is when we know that the foreign data source can't perform joins between the different object mappings we have created for it. Even if the core cluster is a capability, there is still a case when query translation will result in multiple translations taking place; this is when we have joins between mapped types that reside in different data sources. This is fairly easy to understand; consider two distinct data sources that may be completely unaware of each other. How can a query be directed to just one of them that somehow knows about the other data source? In the general case this is not possible, so it's going to be necessary to generate multiple queries, one for each of the data sources. It's then up to Amos II to combine the results of the different queries into a single result set; it's exactly this kind of job a mediator database engine is supposed to do.

Initializers and finalizers

Now we're approaching the final pieces of the Translator API jigsaw puzzle. If we think of the absorbents as being the middle part of a translation, this section will introduce the beginnings and endings of a translation process. *Initializers* and *finalizers* are functions stored on the type of a data source; this makes them general to all instances of that data source. An initializer is called just before translation begins. The only thing the initializer needs to do is to create and return a new, empty accumulator. The finalizer, on the other hand, can have a lot more work to look forward to.

A finalizer is called after all applicable absorbents of a translation have been invoked and it will be passed three arguments: the data source that the translated query should be sent to, the environment (in case we still need it) and the accumulator. At this stage, the accumulator should be stuffed to the brim with everything we need to generate a query in the language of the foreign data source. Once the accumulator has been transformed to a query, which would be a string, what do we do with it? The finalizer clearly shouldn't just return the string directly; the system wouldn't know its meaning. Instead the finalizer should return a function that calls the interface part of the wrapper with the query as its argument. It's these dynamically created functions that will replace all the absorbed functions in a query. Since a complete translation of a query may involve multiple functions being created that need to communicate with each other, these functions don't just produce values but they may receive them as well (this will be shown through an example later). The finalizer, like an absorbent, is allowed to signal failure. In case of failure, all translation to the targeted data source is aborted. It's also at this final stage that we can assign a cost to the function we just produced; this can be of great use to the query optimizer if a good cost metric has been developed.

A SparQL Translator

Now that we've discussed capabilities and absorbents in the abstract, we'll concretize them by using the SparQL translator as an example. The translator is written in ALisp (24) which is a dialect of the Lisp programming language family. ALisp is an eminent choice for writing the translator due to its flexibility and very close integration into the internals of Amos II. Nevertheless, the explanation for

the translator will be given in more abstract higher level terms so we don't get bogged down in programming language level details.

Initial setup

The initial setup that needs to be done is encompassed in one single line from our usage example:

```
set :people = sparql("people","http://url.to/people");
```

A number of separate tasks are taken care of by this call. They are:

- Create a SparQL data source instance. This is what is returned as the result.
- Record the URL of the SparQL endpoint on the data source. We'll need this for later when it's time to send the query.
- Create a core cluster function for the mapped type that will be used to access this data source.
- Record the data source and the default absorbent on the core cluster function. This is part of the practical setup of the translator API.
- Create a mapped type representing an RDF triple for this data source and connect it with the core cluster create above.
- Create a capability for the core cluster on the data source instance we just created and point it to the default absorbent. This is important since we can't translate self-joins without this step. The binding pattern for this capability should always be "fff" to make it maximally general.
- The last step is to register what's known as a *rewrite rule* on the core cluster. The name of this rewriter is always "rewrite-extent". The translator API is built on top of the rewrite system. When using the rewrite system one has to deal directly with the internal representation of AmosQL, something which we are spared when using the translator API. This single incantation is the only vestige of the rewrite system we need to deal with when developing a translator.

The accumulator

The accumulator for the SparQL translator consists mainly of a number of lists that collect the information necessary to construct a valid SparQL query. We have the following lists:

Graph fragment list: Collects internal representations of RDF triples. This will be the foundation for the basic graph pattern (the *where* clause) generated for the SparQL query by the translator.

Output variable list: Collects the variables that the function created by the finalizer is expected to produce values for (the free variables). At least some of these variables will constitute the result.

Input variable list: Collects the variables that will be passed to the function created by the finalizer. These are used to support queries across different data sources and will be explained through an example later on.

Filter list: Collects the filtering capabilities supported by the translator (e.g. arithmetic comparisons). This is the only part of the accumulator that is not for use by the default absorbent.

We also have the following additional data:

Input variable map: Maps variables to unique numbers.

Input variable count: Used to generate new numbers for the variable map.

These two, together with the input variable list, are used to support queries across different data sources and will be explained later.

Translating the core cluster

The most complicated absorbent is probably the absorbent for the core cluster. A birds-eye view of the default absorbent will be offered through pseudo code and a brief explanation, but the easiest way to understand it is probably by following the examples afterwards. The work the finalizer needs to do is quite dependent on the operation of the default absorbent and so will be explained in this section as well.

The default absorbent in pseudo code:

```
gf <- <nil,nil,nil>
for each <arg,entity>
  if constant(arg)
    gf[entity] <- arg
  else if named-variable(arg)
    if bound(arg)
      if datasource(arg) != ds
           and
         arg not in invar-list
        add-invar-mapping(arg,invar-count)
        increase-invar-count
        add-invar(arg)
      gf[entity] <- arg
    else if free(arg)
      bind(arg)
      datasource(arg) <- ds
      add-outvar(arg)
      gf[entity] <- arg
add-graph-fragment(gf)
```

The variable *gf* simply stores a graph fragment (i.e. a triple pattern). The variable *arg* refers to the current variable in the argument list to the core cluster function. The entity tells us if this argument is the subject, predicate, or object, so the syntax *gf[entity]* simply refers to the corresponding place in

the triple. The information connecting *arg* with the appropriate entity is always available since it follows immediately from the position of *arg* in the argument list (the first place being the subject, then predicate, and lastly object). Normally *arg* is a variable, but if it's entered as a constant in the original AmosQL query, the value of this constant will be stored directly in *arg* (so arg will not store a variable in that case). The variable *ds* (which is passed in as an argument to the absorbent) specifies the data source that the core cluster is associated with. For the sake of terseness, no explicit reference is made to the accumulator or environment in the pseudo code.

Translating the core cluster: Example 1

To explain why each of the steps in the absorbent are necessary, we will use two examples. For the examples we assume the foreign data source is a triple store using the FOAF schema to store data about persons. Let's take a look at contestant number one:

```
select o(p1),o(p2) from person p1, person p2
where
  p(p1)='<http://xmlns.com/foaf/0.1/firstname>' and
  p(p2)='<http://xmlns.com/foaf/0.1/lastname>' and
  s(p1)=s(p2);
```

This example shows a self-join used to select the first and last name of all persons in the triple store. Since we introduce two person proxy-objects, we will have two core cluster calls to absorb.

Absorbent, first call:

The first step in every call to the default absorbent is to create an empty triple; we will consider this step as implicit from now on. Next we step over the arguments in order of subject, predicate, and object and take appropriate action. The first variable we see is the subject which is a free variable, so we produce a binding for it by marking it as bound in the environment. We also record its data source (i.e. the data source passed in to the absorbent) in the environment. For our own information, we add the variable to the list of output variables since the function we eventually generate will produce a value for this variable. We also add the variable as the subject to our own triple. The next variable is the predicate. The predicate is a constant so we simply add it as the predicate in our triple. Last is the object which is a free variable; we treat it exactly the same as the subject. Our internal triple now looks like this:

```
<V_sub, "<http://xmlns.com/foaf/0.1/firstname>", V_obj1>
```

For the examples we use descriptive names for the variables. In the actual system, the variables will have internally generated names of the form *_Vx*, where *x* is a non-negative integer. Finally, we add this triple to the graph fragment list. Now we have absorbed this call to the core cluster and the accumulator at this stage looks as follows (untouched lists are not shown):

```
Output variables: (V_sub, V_obj1)
Graph fragments:
  (<V_sub, "<http://xmlns.com/foaf/0.1/firstname>", V_obj1>)
```

Absorbent, second call:

Stepping over the variables, we first see the subject. This time the subject is bound; furthermore, its data source is the same as our own so we don't fall into the complicated looking branch in the pseudo code. The fact that the data sources agree in this way is important because it means that we have a self-join and that we know the subject resides in the data source we are targeting. All we need to do is add this subject to the triple. Nothing new happens in the processing of the predicate and object, so the final stage of the accumulator after both core cluster calls have been absorbed is as follows:

```
Output variables: (V_sub, V_obj1, V_obj2)
Graph fragments:
  (<V_sub, "<http://xmlns.com/foaf/0.1/firstname>", V_obj1>,
   <V_sub, "<http://xmlns.com/foaf/0.1/lastname>", V_obj2>)
```

The astute reader will see that this is in fact all we need to generate a SparQL query. In fact, even the very structure of a SparQL query is visibly present. Now we can move on to the work the finalizer needs to perform which can be divided into two steps: generate a query string, and then, generate a function that uses the string to call the foreign data source.

The generation of the query string is probably the least interesting part of the SparQL translator due to its mechanical and straightforward nature; it's mostly programmer busywork with no particularly difficult theoretical component. The accumulator shown above will result in the following string (indented for clarity):

```
"select ?V_sub ?V_obj1 ?V_obj2
where { ?V_sub <http://xmlns.com/foaf/0.1/firstname> ?V_obj1 .
        ?V_sub <http://xmlns.com/foaf/0.1/lastname>  ?V_obj2 }"
```

We can see that the output variables map exactly to the select clause and the graph fragments map to the where clause. We also have a stroke of luck in that the variable names generated internally by Amos II are already valid names for SparQL query variables, so no name conversion needs to take place.

The actual function the finalizer produces is a bit more interesting to look at. The function itself, both in signature and implementation, is highly dependent on the AmosQL query it was generated from, which means it has little meaning outside the context of the translated query it was made for. Furthermore, every original query made to the foreign data source will create new functions. For these reasons, it's a good idea for the function to be anonymous so it doesn't clutter up the global namespace of functions. Anonymous functions are called *transient functions* in Amos II since they are managed by the system, not the user, and are liable to be garbage collected if they are not referenced from any other object. Transient functions don't have unique identifiers and can't be used, either intentionally or accidentally, in handwritten AmosQL queries (internally, they are referenced by their address, but that doesn't concern us). The other important task the function

must do is converting the vector returned by the SparQL query interface to a tuple. The vector can't be returned directly because it will be treated as a single return value; we have to deconstruct it into a tuple so the system can pick out the separate result variables. One might now wonder why the query interface doesn't return tuples directly. The problem is that when returning a tuple, its size must be known. A SparQL query may return any number of values and we don't know how many until we have collected all the output variables in a translation attempt; only then can we create the result set as a tuple.

According to the above explanation, the function created for this example will have to do something similar to this:

```
create function *transient* () -> <charstring V_sub,
                                   charstring V_obj1,
                                   charstring V_obj2>
as select v[0], v[1], v[2] from vector v
   where v = sparql_query(query,address);
```

Note that this is a visual aid since transient functions can't be created directly in AmosQL (in ALisp it's no problem). The query is simply the string we just generated and the address is retrievable from the data source object (recall that we stored it there when we first created the data source instance). We can also note that the subject is returned here even if it was not explicitly asked for and it's not required to produce the correct result. This doesn't cause any problems since the original AmosQL query only asked for the two objects; it's only the variables that were mapped to these objects that are eventually returned to the user. Going further, we can even say that it would be incorrect to not return the subject. Consider what would happen if the second call to the default absorbent would fail (even if we know it won't, the system can't guess that). In that case the core cluster would be called for the second object and the system would have to perform the join. This would mean that comparing the subjects would be necessary, so in that case, the subject is required to be returned even if it wouldn't be a part of the final result.

Translating the core cluster: Example 2

Now we move on to the second example. This query is similar to the previous one, but with a crucial difference: the two person objects now reside in different data sources.

```
select o(p1),o(p2) from person1 p1, person2 p2
where
  p(p1)='<http://xmlns.com/foaf/0.1/firstname>' and
  p(p2)='<http://xmlns.com/foaf/0.1/lastname>' and
  s(p1)=s(p2);
```

In reality it would probably be insane to store this information across two separate data sources like this; we only use this as an example to explain the general principles of how to translate such a query. Like before, we still have two person objects so we will still have only two core cluster calls to

absorb. Unlike the previous example, we have two objects residing in different data sources, so we will have two separate translations with two separate functions produced by the finalizer.

The first call to the default absorbent is identical to the previous example. The subject and object are asked for with the predicate being a constant, so the accumulator will look like the following:

```
Output variables: (V_sub, V_obj1)

Graph fragments:

  (<V_sub, "<http://xmlns.com/foaf/0.1/firstname>", V_obj1>)
```

The accumulator above is actually completed now and can be finalized. This makes sense since *subject* and *firstname* are all we requested from this data source. We get a query and function similar to this:

```
query = "select ?V_sub ?V_obj1

      where { ?V_sub <http://xmlns.com/foaf/0.1/firstname> ?V_obj }"


create function *transient-1* () -> <charstring V_sub,

                                     charstring V_obj1>

as select v[0], v[1] from vector v

   where v = sparql_query(query, address);
```

The previous translation was completed (i.e. a function was produced) so a new translation with a new accumulator commences for the absorption of the second core cluster call. The first variable we encounter is the subject. This time the subject is bound, but unlike the previous example, it's been bound by a different data source. The fact that the variable has been bound by someone else means that this translation isn't responsible for creating the binding, so we have a join on our hands, but not a join between two types that live in the same data source. Now we are faced with a bit of a conundrum. We can't just incorporate the variable into the query like usual since it has no established meaning for this translation yet. At the same time, the fact that it's bound means that at runtime, the variable will have an actual value when the query is made. What we need is a way to delay the production of the full query until we know what that value is. For now we add this variable to the input variable list, record this variable in the input variable map and increase the input variable counter. The purpose of the map and the counter is to connect each variable bound by another data source to a unique integer (starting at one). Why this is done will be explained soon, but for now, we have made the variable meaningful in the current translation so we add the subject to our internal triple as before. The predicate and the object offer no new complications and are dealt with as previously described. Now the accumulator looks like this (showing the input variable map as a list of tuples):

```
Input variables: (V_sub)

Output variables: (V_obj2)
```

```
Graph fragments:
  (<V_sub, "<http://xmlns.com/foaf/0.1/lastname>", V_obj2>)
Input variable map: (<V_sub, 1>)
Input variable counter: 2
```

The finalizer will work slightly differently with an accumulator such as this. Any variable appearing in the input variable map will not be inserted into the query as is, but will be replaced by a marker that is to be replaced with an actual value when the finalized function is executed. The query string will look as follows:

```
"select ?V_obj2
where { $1 <http://xmlns.com/foaf/0.1/lastname> ?V_obj2 }"
```

A query like this is not very useful in its present form, so what's the trick? The fact that we added *V_sub* to the input variable list (i.e. it is a variable bound by a data source different from the one targeted by the current translation) tells us that this variable will be passed, at runtime, to the finalized function. In particular, the value of the subject will be supplied from the result of the query we translated before this one. The only remaining question is then how do we get these input values into the query. Unfortunately, there's no easy answer to that. Readers familiar with JDBC (25) might know that the JDBC API to SQL has something called prepared statements (26) that can do exactly what we need. SparQL does not have such a facility, so an ad-hoc version was made for this project. The *sparql_param_query* function, in addition to the query and address, also takes a vector of values that will be inserted into the query. The first value in the vector will replace all '$1' markers; the second value will replace all '$2' markers, and so on. We can detect if we need to do a parameterized query simply by checking if there are any variables in the input variable list. The produced function will look like this:

```
create function *transient-2* (charstring V_sub) -> charstring
                                                    V_obj2
as select v[0] from vector v
   where v = sparql_param_query(query,vector(V_sub),address);
```

Now we have successfully translated everything we can from the original query. In case there's confusion as to how the two functions we've produced fit together it might help to look at the following visual aid:

```
select V_obj1, V_obj2 from charstring V_sub,
                           charstring V_obj1,
                           charstring V_obj2
where
  <V_sub, V_obj1> in *transient-1*() and
  V_obj2 in *transient-2*(V_sub);
```

We can also note that when the query is being executed, the *transient-2* function will be invoked once for each subject if the *transient-1* function returns several subjects; this could cause a large number of accesses to be made to the foreign data source since a nested-loop join is used here. The performance can be improved by using different kinds of joins combined with rewrites, but this cannot be handled by the translator API and falls outside the scope of this project.

Translating the core cluster: Some special cases

We're almost done with our exposition on how to absorb the core cluster, but there are a few special cases that can arise that weren't explained above. Let's take care of them now. The first case is when we have a query like the following:

```
select o(pers) from person pers
where p(pers) = '<http://xmlns.com/foaf/0.1/mbox>';
```

Here we're asking for all email addresses in the triple store, regardless of subject. The predicate and object we already know how to handle, but what about the subject? A value is neither sought nor provided for the subject. In the translator, variables such as the subject in the above query will be represented by the symbol '*'. The star variable is a special variable identifier that tells us we can ignore that variable in our query. In the SparQL wrapper the star variable is handled implicitly when we specify we want either constants or named variables (the star variable is not considered as having a unique name). This leaves that part of the triple initialized to its default value (nil, in ALisp). When building the SparQL query we must check for this nil value since every triple pattern must be complete in a SparQL query. What we need to generate is a query variable that is guaranteed to not match anything else in the SparQL query; this is easy since we know the exact format that the material query variables will take. We keep a counter that is increased every time we encounter a nil value and use its current value to generate a dummy query variable. The above example will have the following query string:

```
"select ?obj
where { ?x1 <http://xmlns.com/foaf/0.1/mbox> ?obj }"
```

As an additional example, if we had a query that only asked for all the subjects in the triple store and nothing else, we would get a query string like this:

```
"select ?sub
where { ?sub ?x1 ?x2 }"
```

The other special case we must consider is more problematic and occurs when all the variables in a query are bound. As an example, let's look at the following query that selects all first names matching 'Mikael' in the triple store:

```
select o(pers) from person pers
where
  p(pers) = '<http://xmlns.com/foaf/0.1/firstname>' and
```

```
o(pers) = 'Mikael';
```

We would expect this to return 'Mikael' once for each occurrence in the triple store (this is what would happen if person was a regular type in Amos II and not a mapped type). How do we write this query in SparQL? Naive compliance with the pseudo code produces this query string:

```
"select
where { ?x1 <http://xmlns.com/foaf/0.1/firstname> 'Mikael' }"
```

This is not a valid SparQL query since the select statement must always introduce one or more query variables. We could imagine a hypothetical function produced by the finalizer serving as a predicate returning true for each firstname + Mikael combination appearing in the triple store. Misfortune strikes as we realize that such a query can't be written in SparQL. While SparQL does have the ASK functionality described in the section on SparQL, this only allows us to ask boolean existence queries. So, we can ask if a firstname + Mikael combination appears in the triple store, but we can't ask how many. This is part of a wider issue in that the SparQL standard currently lacks aggregation operators. Queries such as "What is the average age of everyone in the triple store?" or "How many people named 'Mikael' exist in the triple store?" generally can't be asked using only SparQL. Our only recourse is therefore to have the finalizer signal failure if it encounters an empty output variable list and fall back on the core cluster. For the above query this means that everything will be brought in from the triple store and Amos II will then filter out everything that doesn't match our provided predicate and object.

Translating the other capabilities

In this section the other capabilities for the SparQL translator will be explained. These are going to be boolean filtering predicates used for reducing the size of the result set. We have capabilities for arithmetic comparisons and regular expression based literal matching. These are not as theoretically complicated as the default absorbent although they do have some subtle aspects to them. All these capabilities are defined on the type of the SparQL data source making them common to all instances.

Firsts let’s look at the predicate for regular expression based filtering. On the Amos II side this predicate is called “like”, it takes a variable and a pattern used for matching. When defining capabilities such as these it’s usually a good idea to use the full signature of the Amos II function we are targeting to avoid any ambiguities since many of the system functions are overloaded. For this capability the function is named “charstring.charstring.like->boolean”. The binding pattern used is “bb” meaning that both arguments must be provided to the absorbent, it’s not expected to produce any new bindings. Note also that there’s no need to specify an “f” binding for the boolean result; this comes from the practice in Amos II of treating falsity as non-existence in the database, so any result returned that is not filtered by the predicate is by definition also true according to the predicate. For us, this means that we only need to absorb the predicate and use it to restrict the size of the result set; no bindings need to be produced.

Absorbing this predicate currently consists of little more than passing on the values provided to the data source. The only test that needs to be performed is ensuring that the first argument is a variable and that the second argument is a constant. One issue here is that the regular expression language used by Amos II differs from the one used in SparQL; currently the pattern string entered is just

passed on to the foreign data source which breaks transparency but translating between regular expression languages is outside the scope of this project.

Example:

```
select o(pers) from person pers
where
  p(pers) = "<http://xmlns.com/foaf/0.1/firstname>" and
  like(o(pers),"\^A");
```

This query selects all first name beginning with a capital A in the triple store. Once the core cluster and filter have been absorbed the accumulator looks like this:

```
Output variables: (V_obj)
Graph fragments:
  (<NIL, <http://xmlns.com/foaf/0.1/firstname>, V_obj>)
Filter list: ((regex,V_obj,"^A"))
```

The filter list stores all filters absorbed from the query. Each filter is stored internally as a list of the relevant pieces of data since that's the natural way to store things in ALisp. Due to the large variety in potential filters there's no attempt to standardize how the filters are stored except for the head of the list which contains a unique identifier for the type of filter; the tail of the list is filter dependent. During query generation each filter is passed on to its own string generator. The string generated from the above accumulator looks like this (indented for clarity):

```
"select ?V_obj
where
{ ?x1 <http://xmlns.com/foaf/0.1/firstname> ?V_obj .
  FILTER(regex(?V_obj,"^A")) }"
```

The function produced by the finalizer is not affected by the filtering predicates since these predicates do not conjure any new input or output variables.

Next let's look at the comparison operators. These capabilities will cover the functions <, >, <= and >=. We don't include the equality operator = since that one is used to give values as constants. Since all these operations are so similar they will all be handled by the same absorbent. Currently the absorbent also makes the simplifying assumption that one of the arguments should be a constant. For the comparisons we use the function signature "object.object.X->boolean" where X is the operation. The absorbent then checks if the operands are numbers of strings; this gives us both arithmetic and lexical comparisons.

Example:

```
select o(p1) from person p1, person p2
where
```

```
  p(p1) = "<http://xmlns.com/foaf/0.1/name>" and
  p(p2) = "<http://xmlns.com/foaf/0.1/age>" and
  s(p1) = s(p2) and
  o(p2) >= 30 and o(p2) < 40;
```

This query selects the names of everyone in their thirties. After absorption we have the following accumulator:

```
Output variables: (V_sub, V_obj1, V_obj2)
Graph fragments:
  (<V_sub, <http://xmlns.com/foaf/0.1/name>, V_obj1>,
   <V_sub, <http://xmlns.com/foaf/0.1/age>,  V_obj2>)
Filter list: ((cmp,V_obj2,>=,30), (cmp,V_obj2,<,40))
```

The query generated from this accumulator looks as follows (again indented for clarity):

```
"select ?V_sub ?V_obj1 ?V_obj2
where
{ ?V_sub <http://xmlns.com/foaf/0.1/name> ?V_obj1 .
  ?V_sub <http://xmlns.com/foaf/0.1/age>  ?V_obj2 .
  FILTER(?V_obj2 >= 30 && ?V_obj2 < 40) }"
```

Although this query does return a lot of information that we didn't explicitly request, this "surplus value" is not a problem as was explained at the end of example 1 of translating the core cluster.

Performance measurements

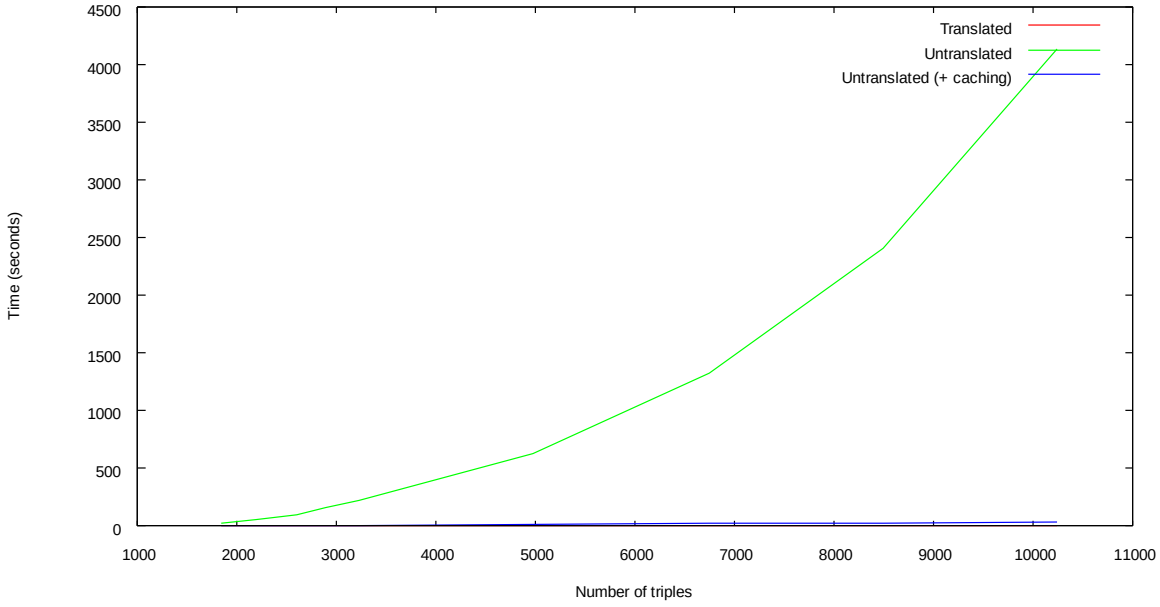
We've stated before that a driving reason for query translation is performance. How big can the performance gains be? The most significant problem we have in a SparQL wrapper is that when writing AmosQL queries based around RDF triples as our only objects, we're going to end up with a lot of self-joins when writing anything but trivial queries. Logically, carrying out a join means forming the Cartesian product of all the requested objects and then restricting the result set based on some join predicates. For objects stored physically in Amos II this can be extensively optimized thanks to specialized algorithms and data structures, but when dealing with a foreign data source the options are more limited. Recall that the only way to communicate with the data source is through the core cluster function which just returns a list of all the attributes of the foreign objects. With this as the only communication primitive, performing a join with a foreign data source means the system will simply try to form all combinations of all the involved objects and later discard all the combinations that were not really requested. We can intuit that this way of forming the result is a process that runs in exponential time over the number of sources to join. When executing queries such as these, the system will repeatedly call the core cluster in a nested-loop-join which ends up in a lot of communication with the data source. One obvious optimization here is to cache the result of the

core cluster when doing untranslated queries to mitigate the communication costs, but this won't solve the problem with time complexity and will require space in Amos II to store the cached rows.

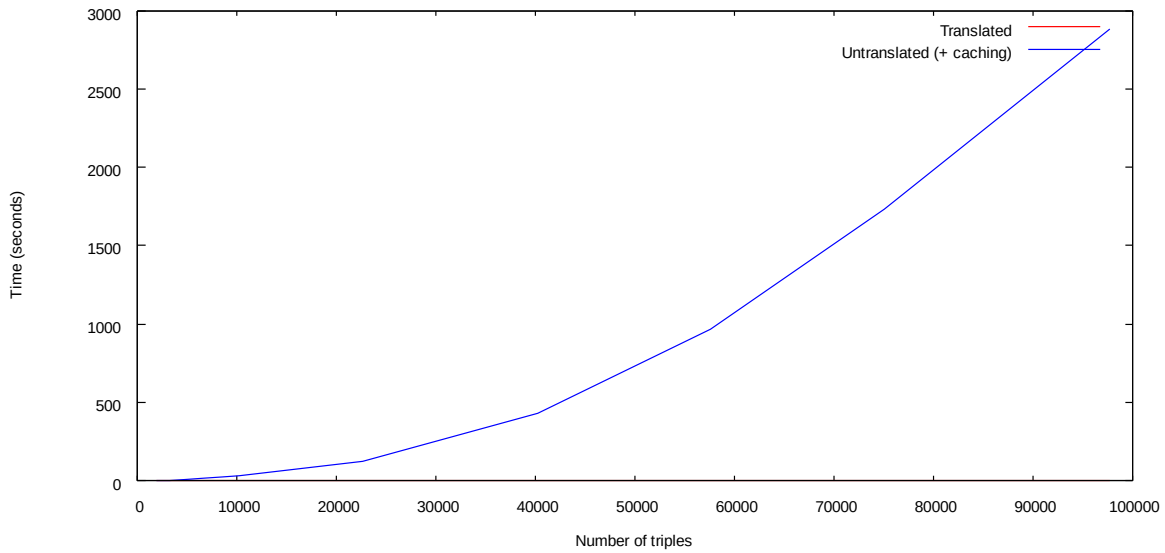
The performance tests will operate on datasets generated by the Berlin SparQL Benchmark; this is a benchmark suite intended to test the performance of SparQL enabled triple stores. Here it will only be used for its ability to generate datasets in a reproducible manner. The queries used will be of the form below and will be sent to three differently enabled data sources: one doing translated queries, one doing untranslated queries but with cached core cluster calls and one doing completely untranslated queries.

```
create function perf_test() -> integer as
  count(select o(ds1),o(ds2) from dataset ds1, dataset ds2
  where p(ds1) = ". . ." and
        p(ds2) = ". . ." and
        s(ds1) = s(ds2));
```

We use the count function in order to suppress the time it would take to just print the results due to the size of the datasets; this doesn't affect the generated queries. Let's look at some time measurements:

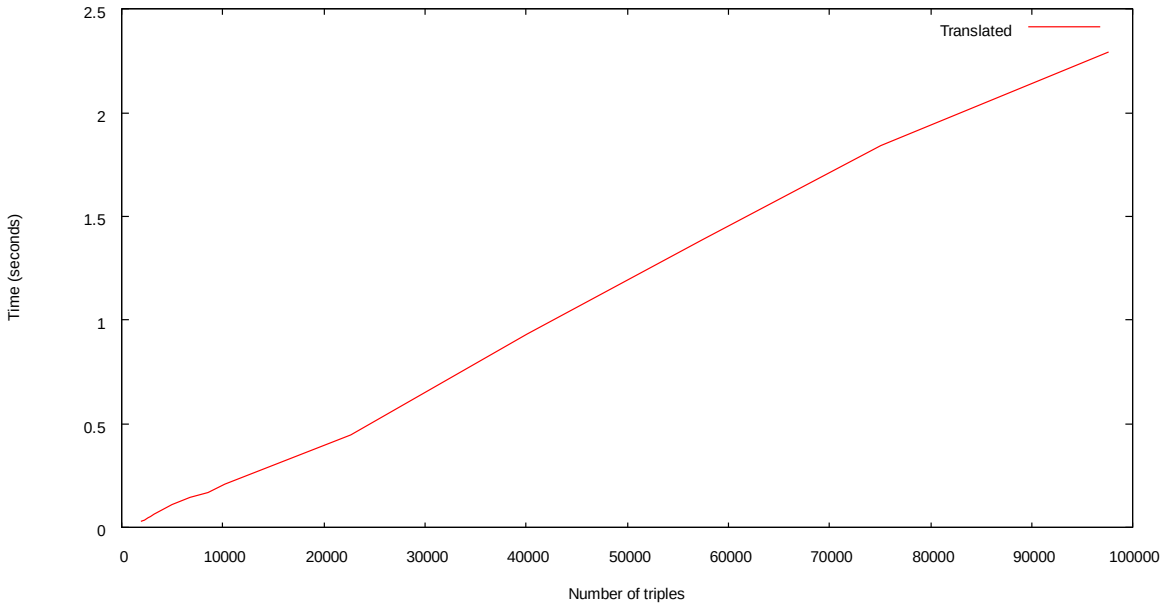


The inefficiency of the untranslated queries is very apparent; for a dataset of around 10000 triples (which isn't that big) it takes around one hour to produce the result. The cached queries perform better, taking almost 30 seconds for the largest dataset. None of the translated queries took longer than a second to complete. Does this mean that untranslated but cached queries are a viable option to translated queries if we don't feel like going through the trouble of writing a translator? No, the cached queries may stave off our performance concerns for a little while by reducing communication costs, but caching alone can't save us from the, in this case, quadratic nature of the underlying algorithm. Taking a peek at slightly larger datasets shows this; here the untranslated queries without caching are not tested:



When the size of the dataset is increased we see that the times of the cached queries begin to skyrocket as well; the largest dataset is about 100000 triples and takes almost 3000 seconds to finish. We can note that for 10000 triples the cached query took almost 30 seconds to finish, so an increase in dataset size of a factor 10 resulted in a time increase of a factor 100; this tells us quite clearly that the untranslated queries, whether cached or not, are experiencing quadratic growth in time.

We can also look at the behavior of only the translated queries; we haven't really seen their result times yet since they have been completely dwarfed by the untranslated queries.



The time to complete the query for the largest dataset (about 100000 triples) is just over 2 seconds. The translated queries are experiencing a steady linear growth. These queries are all being fully executed on the foreign data source, so the time taken is basically just the sum of the time it takes the SparQL data source to run the query, the time it takes to send the result back and the time it takes to parse the result into Amos II objects.

Conclusion and future work

The facilities in Amos II for wrapping and performing high performance queries to a foreign SparQL data source have been explored. We have seen that when dealing with larger datasets, it's essentially mandatory to perform some kind of query translation to reach an acceptable level of performance; doing all the work in the mediator is not feasible. Furthermore, the existing translator API which had previously been used for access to relational databases has been shown to be general enough to achieve access to a different type of data source.

One problem left for the future that has not been satisfactorily solved has to do with types. Charstrings were used to represent the components of a triple which has some limitations. In queries where the object part of a triple is specified as a constant, it's ambiguous if the value is a URI or a literal that simply prints as a URI. The current wrapper treats all object values given as constants as untyped literals. One approach would be to modify the translator and interface to simply require that all literals be quoted twice to remove the ambiguity, but this would be little more than a "hack" as well as being a bit unseemly and easy to forget when writing queries. The proper solution would likely be to incorporate an RDF-aware data type where the components themselves can know what type of value they contain.

Works Cited

1. **W3C.** W3C Semantic Web Activity. *World Wide Web Consortium (W3C).* [Online] 2010. http://www.w3.org/2001/sw/.

2. **W3C.** Resource Description Framework (RDF) Model and Syntax Specification. *World Wide Web Consortium (W3C).* [Online] February 22, 1999. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

3. **W3C**. SPARQL Query Language for RDF. *World Wide Web Consortium (W3C).* [Online] January 15, 2008. http://www.w3.org/TR/rdf-sparql-query/.

4. **Artem Chebotko, Shiyong Lu, and Farshad Fotouhi.** *Semantics preserving SPARQL-to-SQL translation.* 2009, Data & Knowledge Engineering, pp. 973-1000.

5. **W3C.** SparQL Implementation Survey. *World Wide Web Consortium (W3C).* [Online] April 16, 2008. http://www.w3.org/2001/sw/DataAccess/tests/implementations.

6. **Uppsala University, Department of Information Technology.** Amos II. *Amos II.* [Online] 2005. http://user.it.uu.se/~udbl/amos/.

7. **Uppsala University, Department of Information Technology.**. Amos II wrappers. *Amos II.* [Online] 2004. http://user.it.uu.se/~udbl/amos/wrappers.html.

8. **Martin Hansson.** *Wrapping External Data by Query Transformations.* Uppsala : Computer Science Department, Uppsala University, 2003.

9. **W3C.** RDF Primer. *World Wide Web Consortium (W3C).* [Online] February 10, 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

10. **W3C.** Resource Description Framework (RDF): Concepts and Abstract Syntax. *World Wide Web Consortium (W3C).* [Online] February 10, 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

11. **The Internet Society.** Uniform Resource Identifier (URI): Generic Syntax. *The Internet Engineering Task Force.* [Online] January 2005. http://tools.ietf.org/html/rfc3986.

12. **W3C.** XML Schema Part 2: Datatypes Second Edition. *World Wide Web Consortium (W3C).* [Online] October 28, 2004. http://www.w3.org/TR/xmlschema-2/.

13. **W3C.** RDF/XML Syntax Specification (Revised). *World Wide Web Consortium (W3C).* [Online] February 10, 2004. http://www.w3.org/TR/rdf-syntax-grammar/.

14. **W3C.** Turtle - Terse RDF Triple Language. *World Wide Web Consortium (W3C).* [Online] January 14, 2008. http://www.w3.org/TeamSubmission/turtle/.

15. **W3C.** RDF Test Cases. *World Wide Web Consortium (W3C).* [Online] February 10, 2004. http://www.w3.org/TR/rdf-testcases/#ntriples.

16. **Dan Brickley, and Libby Miller.** FOAF Vocabulary Specification 0.98. *xmlns.com.* [Online] August 9, 2010. http://xmlns.com/foaf/spec/.

17. **W3C.** XQuery 1.0 and XPath 2.0 Functions and Operators. *World Wide Web Consortium (W3C).* [Online] January 23, 2007. http://www.w3.org/TR/xpath-functions/.

18. **W3C.** SPARQL Protocol for RDF. *World Wide Web Consortium (W3C).* [Online] January 15, 2008. http://www.w3.org/TR/rdf-sparql-protocol/.

19. **W3C.** SPARQL Query Results XML Format. *World Wide Web Consortium (W3C).* [Online] January 15, 2008. http://www.w3.org/TR/rdf-sparql-XMLres/.

20. **Hewlett-Packard Development Company.** *Jena – A Semantic Web Framework for Java.* [Online] 2009. http://openjena.org/index.html.

21. **Hewlett-Packard Development Company.** *Joseki - A SPARQL Server for Jena.* [Online] 2009. http://www.joseki.org/.

22. **Tore Risch, Vanja Josifovski, and Timour Katchaounov.** *Functional Data Integration in a Distributed Mediator System.* 2004, Functional Approach to Computing with Data, Springer, ISBN 3-540-00375-4.

23. **Staffan Flodin, Martin Hansson, Vanja Josifovski, Timour Katchaounov, Tore Risch, Martin Sköld, and Erik Zeitler** Amos II Release 12 User's Manual. 2010.

24. **Tore Risch.** *ALisp v2 User's Guide.* s.l. : Uppsala Database Laboratory, 2009.

25. **Oracle Inc.** JDBC Overview. *Oracle Technology Network.* [Online] 2010. http://www.oracle.com/technetwork/java/overview-141217.html.

26. **Oracle Inc.** Getting Started with the JDBC API. *Java SE Documentation.* [Online] 2010. http://download.oracle.com/javase/6/docs/technotes/guides/jdbc/getstart/preparedstatement.html.