Uppsala Master's Theses in
Computer Science 306
2007-01-07
ISSN 1100-1836

Processing SparQL Queries in an Object-Oriented Mediator

Yu Cao*

Information Technology
Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

Abstract

RDF is a semantic Web standard that enables Web resources to be described with structure and content. SPARQL is a standard query language for RDF data that provides an easy way to consume the result of queries over a wide range of semantic Web resources. A SPARQL query language parser has been implemented for Amos II, a mediator system that provides view definition capabilities over different kinds of data sources. The SPARQL parser gives the user the ability to query RDF sources using the SPARQL query language. The parser translates SPARQL queries to query statements in the query language of Amos II, AmosQL. The queries contain calls to a wrapper for RDF sources based on a toolkit for manipulating RDF files, Raptor.

Examiner : Tore Risch

* Contact information: wilson_caoyu@yahoo.com

Contents

1. Introduction

Web resources are growing fast day by day, far beyond mainframes' ability to dig out high quality information hidden by the internet. This is largely because each Web resource is created and maintained individually by different users, organizations, companies etc. and is saved in various kinds of formats. Admittedly, search engines like YAHOO and GOOGLE make it possible for us to find interesting information on the internet, but they can only search Web documents as free text. Structured contents, e.g. the name of a book, the name of its author, its publisher and ISBN, are treated as irrelevant information, which largely reduces the merit of Web resource. For instance, by entering the keyword 'Wuthering Heights' to GOOGLE a set of hyperlinks are returned to Web resources whose textual contents include this keyword, while a query like 'Who is the author of 'Wuthering Heights' but when it is first published can not be answered directly. User must manually go through the search result looking for the answers. Moreover, robots that crawl over Web pages to collect useful information for search engines waste their time mostly on parsing Web resources, skipping layout info etc... How to describe properties (meta-data) about Web resources in a uniform format is the problem that RDF [1] solves.

RDF, *Resource Description Framework*, defines a meta-data representation of information about Web resources, such as title, author, modification date of a Web page, copyright and licensing information about a Web document, or the availability schedule for some shared resource[1]. RDF can also be used to represent information that is not directly retrieved on the Web, e.g., information about items available from an online shop. RDF provides a data model that defines how to describe properties of web resources. RDF makes it possible to manipulate Web resources with the help of some RDF oriented query languages, among which SPARQL [2] query language is proposed as standard.

In the implementation of an RDF parser and a SPARQL query processor, we utilize the idea of *mediator* and *wrapper*. A mediator is a functional module in-between data sources and applications, which provides functionalities to hide the data heterogeneity and therefore to enable the application querying over different data sources without knowing their data internal structures. A wrapper is a module that extracts data from different data sources and converts these data to the mediator's internal representation. A mediator also has its own query language to query over the wrapped data sources. Briefly, our idea is to translate SPARQL query statement to the mediator's internal query language to enable queries over RDF sources by an RDF wrapper.

In sum, this work is to provide a SPARQL query processor based on the Amos II database engine [4], a mediator. The work includes to i) develop an RDF wrapper [4] that maps existing RDF statements [3] to data objects in Amos II and ii) develop a SPARQL parser that translates queries to the AmosQL [4] query language of Amos II to retrieve information from the RDF wrapper. The AmosQL queries thus contain calls to the RDF-wrapper.

A previous RDF wrapper [4] was implemented in JAVA language utilizing Jena [5]. The main

purpose to write a new RDF wrapper, referred to as CRDF (C-language RDF parser), is to enhance performance by using the C language. CRDF is based on the Raptor RDF Parser Toolkit [12], which supports RDF statements saved in files of several formats: RDF/XML [6], N-Triples [12], Turtle [12], just to mention a few, and retrieved over a network connection.

2. Background

Before we go deep into the design and implementation of the RDF and SPARQL parsers, we introduce briefly the resource description framework (RDF), mediator technology, and the query language for RDF source as a set of background knowledge to understand the design and implementation issues. For a thorough understanding of RDF, mediators, and query languages over RDF source see [1], [2], [4] and [7].

2.1. Resource Description Framework

This section is designed to provide the basic knowledge required to understand resource description framework. It introduces the basic concepts of RDF and describes how it is presented as graphs as well as in XML syntax.

2.1.1. RDF Triple and RDF Graph

The Resource Description Framework (RDF) is a language for representing information about meta-data in the World Wide Web. RDF is intended for situations in which the meta-data needs to be processed by applications, rather than being only displayed to people [1]. RDF provides a common framework for expressing meta-data, so it can be exchanged between applications without loss of meaning. Not only contents, but also structures or relationships between contents, are represented by RDF. Since it is a common framework, application designers can leverage the availability of common RDF parsers and processing tools. The ability to exchange information between different applications means that the information can be made available to applications other than those for which it was originally created.

Figure 1 RDF triple

Any expression in RDF is a collection of triples, consisting of a *subject*, a *predicate* and an *object* [7], as it is shown in Figure 1. The *subject* is the part that identifies the thing the statement is about. The *predicate* identifies a property of the subject that the statement specifies. The value of a property is called *object*. This can be illustrated by a node and directed-arc diagram, in which each triple is represented as a node-arc-node link. For all RDF triples the subject must be an RDF *URI reference* (see section 2.1.2) or a *blank node* (see section 2.1.4); the predicate must be a URI reference; the object can be either a URI reference, a *literal* (see section 2.1.5) or a blank node. A set of such triples is called an *RDF graph*.

2.1.2. URI and URI Reference

To provide machine-processable statements, RDF requires unique identifiers for identifying a subject, predicate, or object in a statement without any confusion with identifiers used by other users on the internet. Fortunately, the Web provides a general form of identifier for this purpose. That is the *Uniform Resource Identifier* (URI) [1], which is created to refer to anything that needs to be referenced in the statement. RDF uses URIs as the basis of its mechanism for identifying the subjects, predicates, and objects in statements. To be more precise, RDF uses *URI references*. A URI reference is a URI, together with an optional *fragment identifier* at the end. For example, the URI reference 'http://www.example.org/index.html#section2' consists of the URI 'http://www.example.org/index.html' and the fragment identifier 'section2'. RDF defines a resource as anything that is identifiable by a URI reference, so using URI references allow RDF to describe practically anything, and to state relationships between anything as well.

To make this idea more concrete, Figure 2 shows an RDF graph that presents the statement 'there is a Person identified by http://example.org/EM#prof (a subject), whose name (a predicate) is Eric Miller (an object), whose mail address (a predicate) is 'Box 337' (an object), and whose title (a predicate) is 'Dr.' (an object)'.

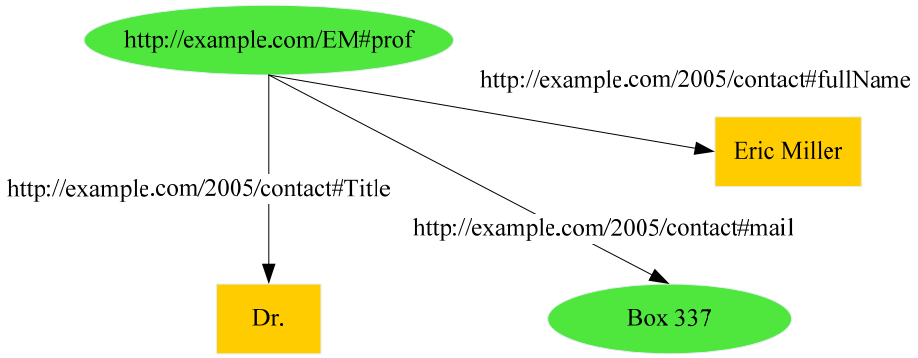


Figure 2 RDF Graph Describing Eric Miller

2.1.3. IRI, Prefixed Names, Relative IRI

An IRI is a generalized form of a URI that can contain non-ASCII characters. IRI is delimited by '<' and '>', while the delimiters are not part of the IRI reference. For example:

```
<http://example.org/staff/staffid/11101022>
```

There are two abbreviations for IRIs, namely *prefixed names* and *relative IRIs*.

- Prefixed names: A prefixed name has a prefix label and a local part, separated by a colon ':'. It is mapped to an IRI by concatenating the local part to the IRI corresponding to the prefix. The prefix label may be an empty string.
- Relative IRIs: Relative IRIs are generated by combining an IRI with a base IRI.

For example, the following fragments are some of the different ways to express the same IRI:

- *<http://example.org/book/book1>*
- *BASE <http://example.org/book/>*
  <book1>
- *PREFIX book: <http://example.org/book/>*
  book:book1

When several resource identifiers share, say the domain name, for instance <*http://a.com/ID*> and <*http://a.com/Salary*>, it is possible to replace the shared part by a predefined label, the prefix label, which denotes part of the resource identifier. For example, there are five URI references denoting five properties of an employee in SPARQL statements as follows:

<http://companyA.com/database/employee#ID>
<http://companyA.com/database/employee#Name>
<http://companyA.com/database/employee#Position>
<http://companyA.com/database/employee#Email>
<http://companyA.com/database/employee#Salary>

All of the five URI references start with ‘http://companyA.com/database/employee#’, so SPARQL query language provides a method to associate a prefix label, say ‘em:’ with this part of the URI reference. The five URI references are converted to an abbreviated version as:

em:ID
em:Name
em:Position
em:Email
em:Salary

In the abbreviated version of the URI references, ‘em:’ is the prefix label, ‘ID’, ‘Name’, ‘Position’, ‘Email’, ‘Salary’ are the local parts.

2.1.4. Blank Nodes

In the real world, for example the mail addresses are usually structured objects. In addition to ‘Box 337’, there can be other information like country, postal code, company name, or department name. For instance the mail address can be ‘Uppsala University, Box 337, SE-751 05 Uppsala, Sweden’. It is not structured if the address is written out as a string to be the object in a triple.

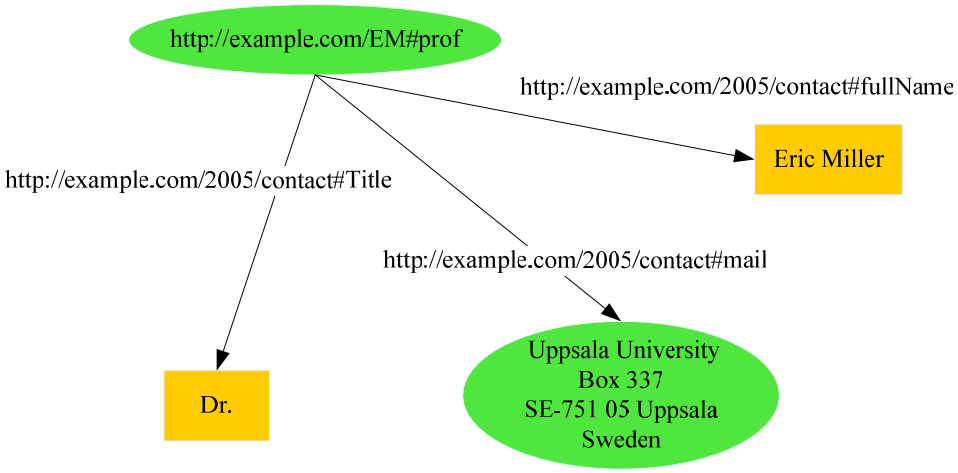


Figure 3 Unstructured Mail Address of Eric Miller

To enable RDF statements to represent structured information, RDF considers aggregated things (like Eric Miller's address) to be described as a resource, and then making statements about that new resource. In the RDF graph, in order to break up the mail address into its component parts, a new node is created to represent the concept of the mail address, with a new URI reference to identify it. However, this way to represent structured information can involve generating numerous ‘internal’ URI references, which is meaningless from outside the RDF graph and therefore will never be referred to, as shown in Figure 4. The solution to these unwanted ‘internal’ URI references in RDF is the *blank node*. A blank node represents internal information that is not accessible from outside the RDF graph. The name of a blank node, the *blank node identifier*, should be unique to one resource in one RDF graph. Given two blank node identifiers, it is possible to determine whether they are representing the same resource in one RDF graph. Therefore, when several RDF graphs are merged, it is necessary to re-allocate the blank node identifiers if two identifiers with the same name refer to two different resources.

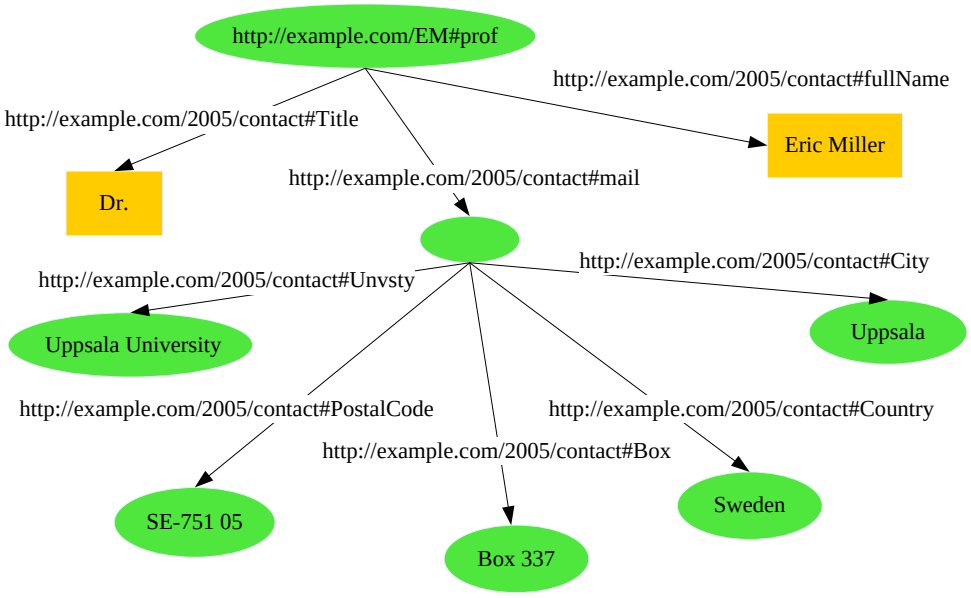


Figure 4 Using a Blank Node

2.1.5. RDF Literals

Literals are used to identify values such as numbers and dates by means of a lexical representation. For instance, integer 23, string '23', double '1.618' are literals in RDF. A literal may be the object of an RDF statement, but cannot be the subject or the predicate.

Literals may be *plain* or *typed* :

- A *plain literal* is a string combined with an optional language tag. This may be used for plain text in a natural language.
- A *typed literal* is a value string combined with a *data type URI*. It denotes the member of the identified data type's value space obtained by applying the lexical-to-value mapping to the literal string.

Examples of literals are:

- A plain literal: *'I know you don't know!'*
- A plain literal with language tag: *'You don't I know you know!'@en*
- A typed literal with a user type: *'xyz'*^^<*http://example.org/datatype*>, in which *'xyz'* is the value string and <*http://example.org/datatype*> is the datatype URI.
- A typed literal integer: *1* It is the same as *'1'*^^*xsd:integer*, in which *'1'* is the value string and *xsd:integer* is the built-in data type URI representing integers.
- A typed literal decimal: *1.3*. It is the same as *'1.3'*^^*xsd:decimal*, in which *'1.3'* is the value string and *xsd:decimal* is the data type URI.
- A typed literal double: *1.0e6*. It is the same as *'1.0e6'*^^*xsd:double*, in which *'1.0e6'* is the value string and *xsd:double* is the data type URI.

For abbreviation, *47* denotes syntax for *'47'*^^*xsd:integer*; *4.7* is the syntax for *'4.7'xsd:decimal; 4.7e10* is the syntax for *'4.7e10'*^^*xsd:double*.

We define that two plain literals are matched, if and only if their lexical values are equal and their data type URIs are equal. For example:

- '*char'* matches '*char*'
- *'char'@en* does NOT match *'char'*
- *10* matches *"10"*^^ *xsd:integer*
- *10* does NOT match *'10'*^^*xsd:double*
- *5* does NOT match *5.0*
- *'1.0e1'*^^*xsd:double* matches *'10.00'*^^*xsd:double*

2.1.6. XML Syntax for RDF

RDF statements are usually stored in an XML-based syntax. The following example is a small piece of RDF in XML-based syntax, RDF/XML, corresponding to the graph in Figure 2:

```
<?xml version="1.0"?>
<rdf:RDF      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#">

  <contact:Person rdf:about="http://www.w3.org/People/EM/contact#me">
    <contact:fullName>Eric Miller</contact:fullName>
    <contact:mail>Box 337</contact:mail>
    <contact:personalTitle>Dr.</contact:personalTitle>
  </contact:Person>

</rdf:RDF>
```

Sometimes it is not convenient to draw RDF graphs when discussing them, so an alternative way of writing down the statements as a set of RDF triples is also used. In the triples notation, each statement in the graph is written as a simple triple of subject, predicate, and object, in that order. For example, the three statements shown in Figure 5 would be written in the triples notation as

```
<http://www.example.org/index.html>
    <http://purl.org/dc/elements/1.1/creator>
        <http://www.example.org/staffid/85740> .

<http://www.example.org/index.html>
    <http://www.example.org/terms/creation-date>
        "August 16, 1999" .

<http://www.example.org/index.html>
    <http://purl.org/dc/elements/1.1/language>
        "en" .
```

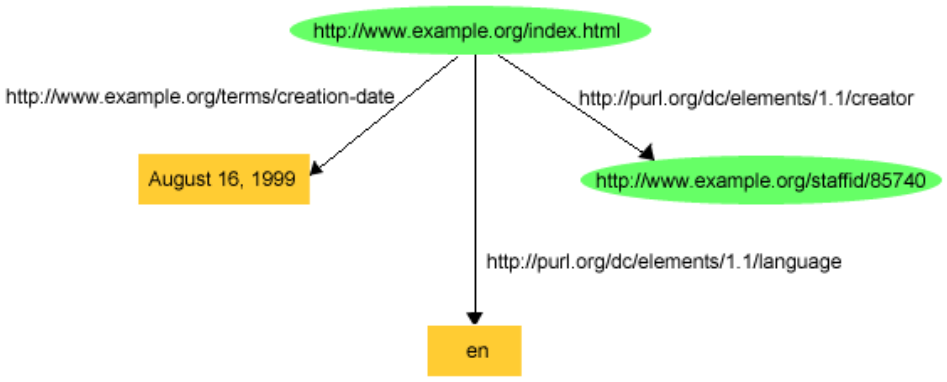


Figure 5 A RDF graph to explain triple notation

2.2. Amos II

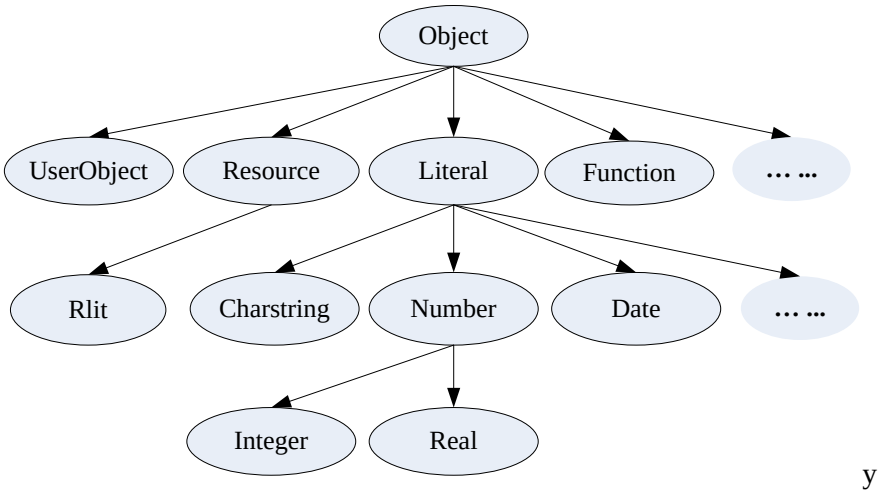
Amos II (Active Mediators Object System) [8] is a distributed mediator system that introduces an intermediate level of software between data sources and applications. Below Encapsulated under Amos II, there might be different kinds of data sources that include, for instance, a DB2 server, Web services, media file repositories, engineering design systems, just to mention some. Information from these external data sources is retrieved by wrappers. Above Amos II, there are applications manipulating data sources through the mediator and being unaware of the existing of wrapped sources. Each Amos II server contains all the traditional database facilities, such as a storage manager, a recovery manager, a transaction manager, and a functional query language named AmosQL [9].

A wrapper is a program module in Amos II that translates data from a particular class of external data sources into the common functional data model used in Amos II [9]. One Amos II mediator may contain one or several wrappers corresponding to various external data sources. It contains both interfaces to external data sources and knowledge of how to efficiently translate and process queries involving accesses to a class of external data sources [9]. In this project, CRDF is a wrapper that translates RDF statements into a stream of Amos II data objects to enable queries over semantic Web resources.

2.2.1. Built-in Data Types

There are a set of Amos II built-in data types to represent a database. Figure 6 shows part of the Amos II built-in types' hierarchy.

- 'Object' is the root type in the hierarchy diagram. It is the supertype of all other types in Amos II, either built-in or user-defined types.
- 'UserObject' is the supertype of the user-defined types (see section 2.2.3 for how to define type).
- 'Resource' is a data type designed to represent Web resources like URIs. A 'Resource' is created in Amos II by calling the built-in function 'r' with its first parameter being set to zero, for instance *r(0, 'http://xxx')* constructs a 'Resource' for the URI *'http://xxx'* in the database inside Amos II. An alternative way to create the same URI in Amos II is to use the function *uri*, e.g. *uri('http://xxx').*
- 'Rlit' is designed to represent literals. A 'Rlit' is created in Amos II by calling the built-in function 'r' with its first parameter being set to lager than zero, for instance *r(1, 'Hello World')* constructs a 'Rlit' for the literal *'Hellow World'* in Amos II.
- 'Charstring' represents character strings, such as *'abcdef'*.
- 'Integer' handles signed numbers, like *47* or *1110*
- 'Real' specifies values that may have fractional parts, such as *1.9* or *6.66e2*



y

Figure 6 Part of the Amos II Built-in Types Hierarchy Chart

Here, we would like to introduce more information regarding the Amos II type ‘Resource’ and ‘Rlit’, because it is these two types that are used to represent URI references and RDF literals in Amos II. The type ‘Resource’ is a built-in data type that has two fields, namely ‘data type ID’ and ‘value string’. ‘Data type ID’ indicates the RDF data type identification for a ‘Resource’. The ‘Value string’ stores the content of a ‘Resource’. ‘Rlit’ is a subtype of ‘Resource’ with its data type ID larger than zero. We can create a ‘Resource’ by invoking Amos II type function *r(integer datatypeID, charstring valuestring)*, for example:

```
r(0, ‘http://a.com/database’);
r(1, ‘http://a.com/database’);
r(3, ‘‘10’^^xsd:integer’);
```

These three AmosQL statements all create instance of Amos II type ‘Resource’s. Moreover, the latter two expression creates Amos II type ‘Rlit’, because they are created with their data type ID larger than zero. Section 4.1 explains how types ‘Resource’ and ‘Rlit’ represent URI references and RDF literals. For creating a URI reference, we can also use the AmosQL function ‘uri’, which invokes AmosQL function ‘r’ with an input of zero as its data type ID. For example,

`r(0, ‘http://a.com/database’);` is equal to

```
uri(‘http://a.com/database’);
```

2.2.2. Simple Select Statements

A simple query expressed as a *select statement* in Amos II has syntax as following:

‘select’ *variablecomma_list*
from_clause
where_clause;

variablecomma_list := variable-name [,variable-name]*
from_clause := ‘from’ type variable-name [,type variable-name]*

where_clause := ‘where’ predicate-expression[①]

For example, a simple select statement can be like:

```
AmosQL query 1
    select name, population
    from Charstring name, Integer population, Charstring City
    where CityName(City) = <name, population>;
```

After keyword ‘from’ in the statement, there are declarations of local variables used in the query. Any variable used in the select statement must be declared in this part; otherwise, the system will report an error. The system binds values to all variables in the query and generates the results by returning the values bound to the variables in variablecomma_list. There can be none, one, or several results to one query. The predicate-expression provides a set of criteria for searching the results.

2.2.3. Types and Functions

The data model of Amos II contains objects, types, and functions. It supports object-oriented representations like instantiation, inheritance, dynamic-binding etc. To manipulate information from stored data, Amos II uses *types*, *functions*, and *procedures* [10].

User can define their own types in Amos II for particular purposes. These user-defined types may have properties. For example, the following statements create a type named ‘Person’ with a property ‘Name’ and an inherited type ‘Student’ with additional properties ‘class’ and ‘su’.

```
create type Person properties (name Charstring);
create type Supervisor under Person;
create type Student under Person properties (class Charstring, su
    Supervisor);
```

Once a type is created, it is possible to create an instance of the type as:

```
create student(name) instances :adam ('Adam');
```

which generates an instance of type ‘student’ whose name is ‘Adam’? Later, ‘:adam’ is the identifier used to access the properties of student ‘Adam’.

There are four types of functions and one type of procedures in Amos II, namely

- *Stored function*, explicitly storing information in the local Amos II. For example,

  ```
  create function employee(Charstring)->Employee as stored.
  ```

- *Derived function*, defined by a single query statement. Amos II optimizes derived function before it is executed. For example,

[①] For more information of predicate-expression, please read section 2.4.3 of [4]

```
create function count_employee(Charstring company)->Integer as
          select count(employee(company));
```

- *Foreign functions* provide low level JAVA, C/C++, or Lisp language interfaces to wrapped external systems. For example,

```
create function sqrt(Real x) -> Real as foreign ‘sqrtbf’;
```

- *Overloaded functions* have different implementations depending on the argument types in a function call.[4]
- *Stored procedures*, defined by AmosQL statements. For example,

```
create function hello_world (Charstring r)->Boolean as
begin
  print(‘Hello world’); print(r); result true;
end;
```

Function names may be overloaded, i.e., functions having the same name may be defined differently for different argument types or result types. This allows overloaded functions to apply to several different object types [4]. The query compiler resolves the correct resolvent to apply based on the types of the arguments; the type of the result is not considered.[4] For example:

```
create type Vehicle;
create type Car under Vehicle properties (maxSpd Integer);
create type Truck under Vehicle properties (capacity Integer);
create function feature(Car o)->Integer as select maxSpd(o);
create function feature(Truck o)->Integer as select capacity(o);
create Truck(capacity) instances :BigBigGo(80);
```

To ask for the feature of the truck :BigBigGo, we input ‘feature(:BigBigGo)’ and Amos II will find the correct function ‘feature’ to invoke and return the answer ‘80’.

2.3. Query Languages for RDF

The increasing number of implementations of RDF after it was firstly recommended in 1998 lead to a requirement for an RDF query language. Up to now, there are several different RDF query languages for instance SQL-like RDQL [11] and XPath-like Versa[14], OQL-like RQL [12], and Datalog-like QEL [13]. RDQL became popular and easy to understand because of its SQL-like syntax. Nevertheless, without a standard, the implementations of RDQL are similar but have some divergence and extensions [11]. Therefore, in Feb 2004, the W3C formed the RDF Data Access Working Group (DAWG), who published a working draft of design: *SPARQL Query Language for RDF* on 12, Oct, 2004. The latest candidate recommendation specification of SPARQL was released on 6, Apr, 2006.

SPARQL is designed to satisfy the following demands for an RDF query language [11]:

- Supports all of the RDF model
- Knows about RDF graphs and RDF triples
- Can handle RDF's semi-structured data
- Supports operations on RDF graphs
- Enables higher-level application development
- Enables cross-language, cross-platform development

The SPARQL query language is based on matching graph patterns, which is like an RDF triple pattern, but with the possibility of a variable instead of an RDF term in the subject, predicate or object positions. Desired information is retrieved by binding values to variables in the query graph pattern against one or several RDF graph. SPARQL provides facilities to extract information in the form of URIs, blank nodes, plain and typed literals, and to extract RDF sub graphs and construct new RDF graphs based on information in the queried graphs [2]. For instance, the following query in Figure 7, in which '?name' and '?email' are variables, would have one solution in the graph in Figure 2. This query graph could be expressed in SPARQL syntax as

```
SPARQL query 1
    PREFIX w3Contact: < http://www.w3.org/People/EM/contact#>
    SELECT ?name, ?mail
    WHERE
    {
        w3Contact:me w3Contact:fullname ?name.
        w3Contact:me w3Contact:mail ?mail.
    }
    ORDER BY ?name ?mail
    LIMIT 5
    OFFSET 10
```

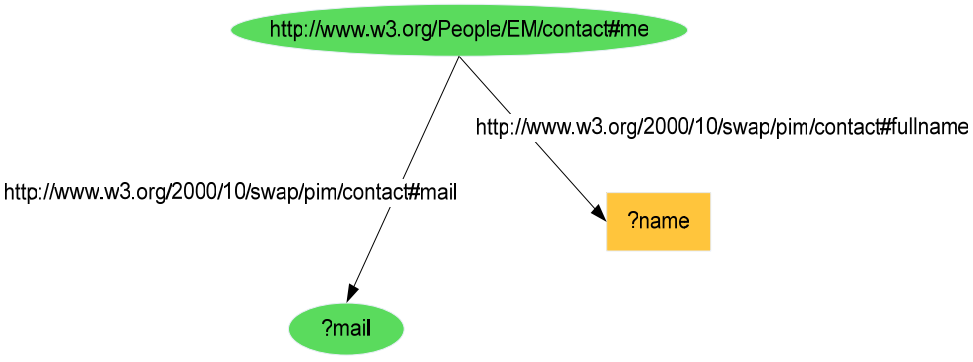


Figure 7 RDF query graph

In the query statement, keyword 'ORDER BY' specifies a sorted result list. The results should be increasingly sorted first on the value of variable 'name' and second, on the values of variable 'mail', in case the values of variable 'name' of two results are same. Keyword 'LIMIT' specifies a limitation on the number of results, say in this example, the number of results should be less than or equal to 5. The keyword 'OFFSET' causes the results generated to start after the specified number of results. As shown in the example, the first 10 results

should be ignored and the 11th become the first result returned when there are more than 10 results to the query, otherwise no result is returned.

The query example above illustrates a typical structure of a SPARQL query statement that includes four parts:

- *Prolog*. The PREFIX keyword associates a prefix label with an IRI.
- *Variable List.* It follows the 'SELECT' keyword, to select variables appearing later in the *Where Clause*. These variable bindings are the result tuples from the query.
- *Where Clause.* Starting with the 'WHERE' keyword, the *Where Clause* is the most complex part of a SPARQL query. It contains a set of RDF triples specifying a restriction on the results to return from the query. In addition to that, there are *Value Constraints* and *Optional Graph Pattern*[①].
- *Solution Modifier.* This part provides clauses that modify the result to a query. Operations on result include sorting, limiting, and offsetting.

Generally, SPARQL queries retrieve information in three phases: i) the *pattern matching* binds values to variables based on matching RDF graph patterns to generate a solutions set, ii) the *value constraints* eliminate solutions violating allowable bindings of variables to RDF term from a solutions set, and iii) the *solution modification* can contain specifications of how to sort the results, limit the number of the results, or offset the first N results.

Our task of converting a SPARQL query statements to AmosQL query statements covers how to translate clauses in the prolog, the where clause, and the solution modifier. In addition, other issues, for example typed-literals [14], built-in calls, effective Boolean value (EBV)[2], operator mapping etc. are also important for our fully-functional SPARQL parser design. Optional graph patterns, which imports unbound variable in the result, are described in the specification of the SPARQL query language, but are excluded in this implementation.

3. The SPARAMOS System

3.1. Architecture

SPARAMOS is an implementation of the SPARQL query language that has two parts: *CRDF*, a wrapper that imports RDF document into Amos II, and the *SparQL Parser* that translates SPARQL queries to AmosQL. This allows SPARQL queries over semantic Web data sources. The SparQL Parser works with the help of CRDF, while CRDF is a general wrapper of RDF sources and is independent of SparQL. Amos II is the kernel of SPARAMOS that manages storage, optimizes queries, generates execution plans, binds values to variables etc. Figure 8 shows the architecture of the SPARAMOS system.

① *Value Constraint* and *Optional Graph Pattern* will be explained in latter section.

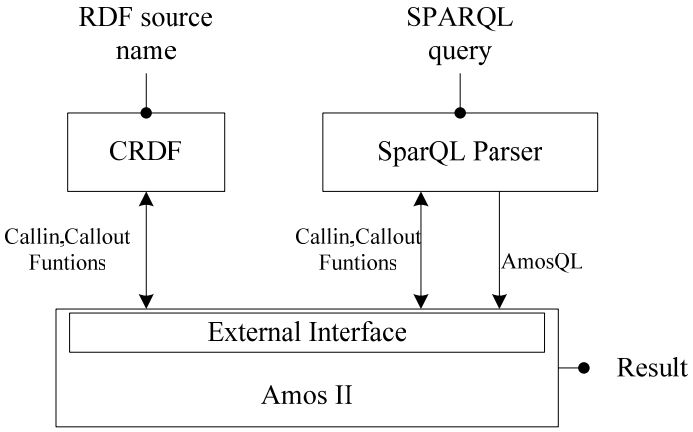


Figure 8 Architecture of SPARAMOS

The SparQL Parser accepts a SPARQL query statement and translates it to a string containing an AmosQL 'SELECT' query statement. The string is then sent to Amos II for execution as an argument for the Amos II built-in function 'eval' that evaluates a query string. The result from 'eval' is a stream of result tuples represented as vectors. The translated AmosQL statements contain calls to the CRDF wrapper to access RDF triples from the queried sources. The matched RDF graph pattern is found in these triples. The retrieved result triples may need to be post-processed by solution modifications in the SPARQL query. If the SPARQL query call for a sorted result, for instance, the built-in AmosQL function 'sortbagby' is invoked to sort the result.

3.2. The CRDF Wrapper

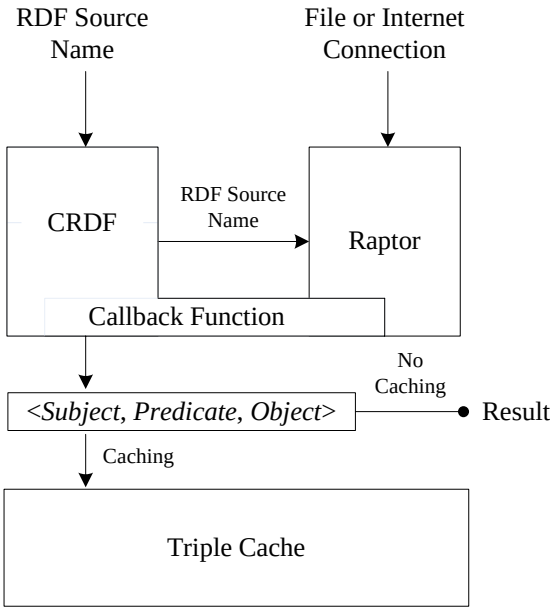


Figure 9 Architecture of CRDF

CRDF uses the Raptor RDF Parser Toolkit [12] to parse RDF terms including data-typed literals and XML literals. Raptor is a free software/Open Source C library that provides a set

of parsers that generate RDF triples by parsing various RDF formats. The supported parsing syntaxes are RDF/XML, N-Triples, Turtle, RSS tag soup including Atom 1.0 and 0.3, GRDDL for XHTML and XML. The supported RDF syntaxes are RDF/XML (regular, and abbreviated), N-Triples, RSS 1.0, Atom 1.0 and Adobe XMP. Whenever generating an RDF triple, Raptor invokes the registered *callback function* to retrieve information of subject, predicate and object, which are converted to 3-tuple <subject, predicate, object> that Amos II recognizes. The 3-tuple is emitted as a stream element to Amos II. In case there are needs to materialize (cache) the 3-tuple in the database, the callback function associates the 3-tuple with its RDF source name and stores this information by invoking a stored function. This is called is the *cache mechanism* of CRDF. Sections 4.2 and 4.3 explain the cache mechanism and its advantages and disadvantages.

By repeated invocations of the callback function, CRDF retrieves all of the RDF triples in an RDF document; the RDF triples are converted to AmosQL format and emitted or cached. As a result, for a given an RDF document we can ask Amos II for all of its RDF triples.

3.3. The SparQL Parser

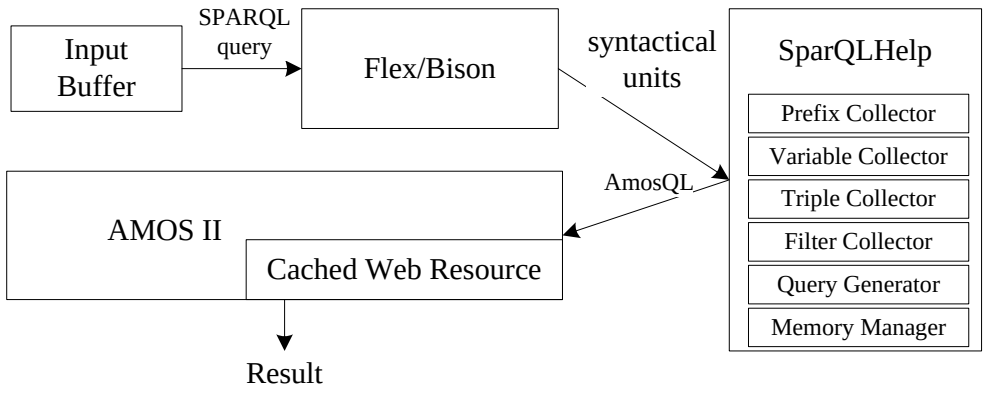


Figure 10 Architecture of SparQL Parser

The architecture of the SparQL Parser is shown in Figure 10. The SparQL Parser accepts SPARQL query statements from a file or from standard input. It is first saved it in an *input buffer*. Secondly, *Flex/Bison* uses the SPARQL grammar to recognize lexical patterns and parse the SPARQL query statement. The SparQL Parser stores data representations of variables and prefixes. Then it associates local parts with prefix labels, generates AmosQL statements, etc.. The data representations are provided by the module *SparQLHelp* that uses them to produce an AmosQL query over RDF resources.

SparQLHelp contains several modules for different purposes:

- The *Prefix Collector* saves information of the prefix label associated with an IRI.
- The *Variable Collector* stores names of the variables appearing in a parsed SPARQL statement.
- The *Triple Collector* collects triple patterns <*subject*, *predicate*, *object*> in the SPARQL statement.

- The *Filter Collector* stores information to restrict solutions that constrain the bindings of variables to RDF terms.
- The *Query Generator* generates an AmosQL statement from collected information.
- The *Memory Manager*, handles memory allocation.

4. Implementation of CRDF

4.1. Data Types of Subject, Predicate, and Object

An important issue that must be decided before starting to implement CRDF is which Amos II data types are used to represent an RDF triple. When triples are return by the invoked callback function as discussed in section 3.2, CRDF must decide to which Amos II data types the subject, predicate, and object is going to be converted to so that the triples can be represented in Amos II. There are only three types of data involved in presenting an RDF triple: URI references, blank nodes, and literals (see section 2.1). They are discussed in the following two sections.

4.1.1. URI Reference and Blank Nodes

Section 2.1.2 and Section 2.1.4 introduced what is a URI references and blank nodes of RDF. A URI reference identifies resources accessible from outside the RDF document. A blank node identifies ‘internal’ resources that are meaningless outside the RDF document. Blank nodes may share same names in several RDF documents, but they identify not necessary the same resource, say ‘_:node1000’ represents mail address in one RDF document and represents a 3D model in another RDF document. Inasmuch URI references and blank nodes share a common property — identifying resources, the Amos II type ‘Resource’ is the proper data type to represent both of them.

To convert URI references and blank nodes to Amos II type ‘Resource’, we use the internal Amos II C function ‘new_R’ in CRDF. Its areguments are i) a data type ID that indicates creating a Amos II type ‘Resource’; here zero identies the ‘Resource’ as a URI reference. It is also possible to indicate creating a Amos II object of type ‘Rlit’ by setting the data type ID argument lager than zero.; ii) a ‘Charstring’C string to indentify the value of the resource. When Raptor invokes the callback function, a URI reference or blank node in the triple is saved in a C/C++ ‘char string’ format terminated with ‘\0’. For example,
URI reference

<http://a.com/vehicletype> is created by calling in C

```
new_R(0, ‘http://a.com/vehicletype’);
```

or by AmosQL

```
uri(‘http://a.com/vehicletype’); or
r(0, ‘http://a.com/vehicletype’);
```

Blank node

'_:node1110' is created by calling in C

```
new_R(0, '_:node1110');
```

Calling 'new_R' with its first parameter not being zero creates RDF literals, which is going to be introduced in the next section.

4.1.2. Literals in CRDF

Section 2.1.5 introduced that there are two types of literals in RDF documents: plain literals with optionally language tag, and typed literals. At first glance, the proper mapping of literals to Amos II types should be as follows:

- Plain literals are converted to Charstring, like 'Hello' ==> 'Hello'
- Typed literal strings are converted to Charstring, like 'Hello'^^xsd:string ==> 'Hello'
- Typed literal integers are converted to Integer, like '1'^^xsd:integer ==> 1
- Typed literal integers are converted to Integer, like '01'^^xsd:integer ==> 1
- Typed literal doubles are converted to Real, like '2.9'^^xsd:double ==> 2.9
- Typed literal decimals are converted to Real, like '2.9'^^xsd:decimal ==> 2.9

This mapping method has a serious problem: the value string and the RDF type information of typed literals are lost permanently once they are converted to an Amos II literal. As it is shown in the previous example, both ''2.9'^^xsd:double' and ''2.9'^^xsd:decimal' are interpreted as 2.9 of Amos II type 'Real'; plain literal 'Hello' and typed literal ''Hello'^^xsd:string' share the same mapping result 'Hello' of Amos II type 'Charstring'. Neither of their type information is preserved. In addition, once the typed literal ''01'^^xsd:integer' is converted to 1 of Amos II type 'Integer', its value string '01' is lost permanently. In sum, mapping typed literal to Amos II literal types makes the reversed mapping operation underterministic.

To avoid the problem, literals in CRDF are stored as Amos II type 'Rlit' by invoking the C function 'new_R' or AmosQL fucntion 'r' with its data type ID argument larger than zero. The data type ID indicates the data type URI of a plain or typed literal as following:

Data type URI	Data type ID
N/A (Plain literal)	1
http://www.w3.org/2001/XMLSchema#integer	2
http://www.w3.org/2001/XMLSchema#decimal	3
http://www.w3.org/2001/XMLSchema#double	4
http://www.w3.org/2001/XMLSchema#boolean	5
http://www.w3.org/2001/XMLSchema#string	6
User-defined data type literal	from 7

Data type IDs 1 to 6 are preserved for known typed literals, and from 7 the data type IDs are

for user-defined data type literals. There is a stored function, who is defined as:

create function RDFLiteralID(Charstring)->Integer key as stored;

It keeps tracking how data type URIs are mapped to corresponding data type IDs, for example an AmosQL asking the data type ID for data type URI 'http://www.w3.org/2001/XMLSchema#string'

*RDFLiteralID('*http://www.w3.org/2001/XMLSchema#string*')*

returns 6.

Following are some examples showing how known typed literals are converted to Amos II type 'Resource':

- 'Hello' is converted as *new_R(1, 'Hello')*
- 'Hello'^^xsd:string is converted as *new_R(6, 'Hello')*
- '1'^^xsd:integer is converted as *new_R (2, '1')*
- '01'^^xsd:integer is converted as *new_R (2, '01')*
- '2.9'^^xsd:double is converted as *new_R(4, '2. 9')*
- '2.9'^^xsd:decimal is converted as *new_R (3, '2. 9')*

In addition, to create a user-defined typed literals for a given data type URI, CRDF firtly tries to find an exiting mapping from this URI to its corresponding data type ID; if not CRDF inserts a new mapping to generate a corresponding data type ID and creates the 'Rlit' with the newly created data type ID. For example,

- 'ABC'^^<http://www.abc.com> can be converted as *new_R(7, 'ABC')*
- 'CNN'^^<http://www.cnn.com> can be converted as *new_R(8, 'CNN')*

Last issue left open is how plain literal with a language tag is represented in Amos II. Inasmuch plain literal with language tag is form of a plain literal, CRDF represents it with an Amos II type 'Rlit' with data type ID equal to one. The language tag becomes part of the content of the 'Rlit', for instance:

- 'ABC'@en is represented as *new_R(1, "ABC'@en')*
- 'CNN'@en-gb is represented as *new_R(1, "CNN'@en-gb')*

4.2. Triple Cache

CRDF works as a wrapper that parses RDF source files by building an RDF triple graph over to be queried in SPARQL. It is predefined how many times CRDF goes through the RDF source to finds the result to a SPARQL query. If the RDF source is either a big file or a narrowband internet connection, parsing the source becomes a critical bottleneck for the performance of the SparQL Parser. Therefore, we introduce a *triple cache* mechanism to CRDF. When an RDF source is parsed for the first time, CRDF retrieves and stores the RDF triples in Amos II. The next time the same RDF source is used, the cached information is retrieved instead of reparsing the RDF source again. In the implementation of the SparQL

Parser, there is an argument in the function that specifies whether the queried RDF source should be cached or not. (Details are given in section 5)

The triple cache makes it possible to index the subject, the predicate, or the object of the triples in one cached RDF graph, which speeds up the execution of SPARQL query statement. However, the triple cache brings an update mechanism issue to consideration. This is discussed in section 4.3.

4.3. Summary

In CRDF, URI references and blank nodes are stored as Amos II objects of type ‘Resource’, a built-in type to represent Web resources. RDF literals, both plain literals and typed literals, are stored as Amos II type ‘Rlit’. For example, the RDF triple

```
<http://a.org/index.html>
   <http://a.org/terms/creation-date>
      ‘August 16, 1999’ .
```

is stored in Amos II as a 3-tuple <Resource, Resource, Rlit>

```
<uri(‘http://a.org/index.html’),
   uri(‘http://a.org/terms/creation-date’),
      r(1, ‘August 16, 1999’)>
```

Table 1 gives a summary of how RDF triples are mapped to Amos II built-in data types. The data type of subject and predicate in an RDF triple is always an object of Amos II type ‘Resource’, while the object of a triple can be either ‘Resource’ or ‘Rlit’ depending on whether it is a URI, blank node, or a literal. As a result, the Amos II data type of components in RDF triples are ‘Resource’, for subject and predicate, and for object it is ‘Resource’ or ‘Rlit’.

Table 1 Component Types and Corresponding Data Types

Component	**Component Type**	**Amos II Data Type**
Subject	URI reference	‘Resource’
	blank node	‘Resource’
Predicate	URI reference	‘Resource’
object	URI reference	‘Resource’
	blank node	‘Resource’
	Literal	‘Rlit’

The implementation of CRDF becomes easy with the help of Raptor. CRDF first registers a callback function ‘_rdf_triple_’ in Raptor, which is invoked when Raptor has found an RDF triple while parsing the RDF source. In the callback function, an RDF triple <subject,

predicate, object> is converted to a 3-tuple in format <Resource, Resource, Resource>. The triples collected from a certain RDF source are associated with the source name. The function that reads RDF triples into Amos II has the following form:

```
parseRDF(Charstring src)-><Resource, Resource, Resource>
parseRDF_cache(Charstring src)->Boolean as foreign "parseRDF_cache"
```

For example:

```
parseRDF('c:/test.rdf'); //for not caching RDF source
parseRDF_cache('c:/test.rdf'); //for caching RDF source
```

The function 'parseRDF' emits a bag all of the triples in the file 'c:/test.rdf' in a format of 3-tuple as <Resource, Resource, Resource>. By contrst, when caching RDF source is required, the function 'parseRDF_cache' caches the triples in the RDF source file 'c:/test.rdf' in a system function 'rdf_triple_cache', defined as:

```
create function rdf_triple_cache(Charstring)->Bag of
                        <Resource, Resource, Resource> as stored;
```

For example, say file 'c:/test.rdf' contains four triples:

```
<http://example.org/employee>
       <http://a.com/ID>
           <http://example.org/ID1110>.
<http://example.org/ID1110>
       <http://a.com/name>
              'Yu Cao'.
<http://example.org/ID1110>
       <http://a.com/sex>
              <http://a.com/sex#male>.
<http://example.org/ID1110>
       <http://a.com/salary>
              0.
```

We read this file by command:

```
parseRDF('c:/test.rdf');
```

or

```
parseRDF_cache('c:/test.rdf');
```

Function 'parseRDF' emits all of the four 3-tuples immediately as:

```
<uri(http://example.org/employee),
       uri(http://a.com/ID),
           uri(http://example.org/ID1110)>
<uri(http://example.org/ID1110),
       uri(http://a.com/name),
              r(1, 'Yu Cao')>
```

```
<uri(http://example.org/ID1110),
       uri(http://a.com/sex),
              uri(http://a.com/sex#male)>
<uri(http://example.org/ID1110>),
       uri(<http://a.com/salary>),
             r(2, '0')>
```

Function 'parseRDF_cache' calls foreign function 'parseRDF_cache' to caches the RDF triple associated with file name 'c:/test.rdf', emits true if RDF source is successfully parsed and cached. We can retrieve cached RDF triples by calling

```
rdf_triple_cache('c:/test.rdf');
```

Now, we can query whether there is an employee named 'Yu Cao',

```
AmosQL query 2
    select parseRDF('c:/test.rdf')=<id,
                                uri(http://a.com/name), r(1,'Yu Cao')>
    from Resource id
    where parseRDF('c:/test.rdf')=< uri(http://example.org/employee),
                                uri(http://a.com/ID), id>;
```

The caching mechanism raises an updating problem that CRDF does not solve. When CRDF caches an RDF source, it generates a restructured copy of the original RDF source. The SparQL Parser executes AmosQL translated from SPARQL query over the copy, not the original RDF source. Nobody notifies CRDF when the original RDF source is changed and where the changes is. The SparQL Parser takes a risk on querying over out of date RDF data. Currently, to update the cached RDF source, we have to firstly remove the cached copy by calling AmosQL command 'remove' as:

```
remove rdf_triple_cache("c:/test.rdf")=<s,p,o>
from Resource s, Resource p, Object o
where rdf_triple_cache("c:/test.rdf")=<s,p,o>;
```

The next time we ask for the triples in RDF source 'c:/test.rdf', CRDF will reparse the RDF source and therefore update the RDF source. If the RDF source is seldomly used, or frequently updated, the function 'parseRDF' is an alternative that does not cache the RDF source and therefore does not introduce the updating problem.

5. Implementation of the SparQL Parser

This section includes explanations of how the SparQL Parser translates SPARQL query statements to AmosQL; how the SparQL Parser finds the solutions to SPARQL queries, and in which format they are emitted. The SparQL Parser achieves these purposes with the help of the CRDF wrapper and the statically linked RDF source parser library. The CRDF wrapper provides two interfaces to access RDF sources: 'parseRDF' and 'parseRDF_cache', whose functions have been introduced in section 4.3. To retrieve the triples in an RDF source, the SparQL Parser defines two functions, namely 'rdf_tr' and 'rdf_tr_c', corresponding to no

cached and cached parsing of an RDF source. They are defined as follows:

```
create function rdf_tr(Charstring src)-><Resource,Resource,Resource> as
    select parseRDF(src);

create function rdf_tr_c(Charstring src)-><Resource,Resource,Resource>
as begin
    if notany(rdf_triple_cache(src)) then
        parseRDF_cache(src);

    result rdf_triple_cache(src);
   end;
```

In cases when there are no needs to cache the RDF source, the SparQL Parser invokes function ‘rdf_tr’ to parse the source. Otherwise, when caching RDF source is required, function ‘rdf_tr_c’ is used instead. Function ‘rdf_tr_c’ checks whether an RDF source has been parsed and cached before it starts to parse the source. If the RDF source is parsed for the first time, fucntion ‘rdf_tr_c’ invokes CRDF function ‘parseRDF_cache’ to retrieve and cache the RDF triples in Amos II. For example, statements to retrieve triples in RDF source ‘c:/test.rdf’ with caching and without caching are like:

```
rdf_tr_c(‘c:/test.rdf’); // caching
rdf_tr(‘c:/test.rdf’);  // no caching
```

In the examples in the rest of section 5, function ‘rdf_tr’ is used, but the examples also work correctly if function ‘rdf_tr’ is replaced by function with ‘rdf_tr_c’. The only difference is that, in the later situation, RDF sources are cached. There is an argument in the function that accepts SPARQL query statement to select whether the queried RDF source should be cached or not, which will be explain in section 5.10.

5.1. Prefix Collector

The prefix collector is designed to collect prefixed names and relative IRIs.

In SPARQL statements, prefixed names are declared as:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
```

Base IRI for relative IRI is declared as:

```
BASE <http://example.org/>
```

‘PREFIX’ is the keyword indicating a definition of prefix label, ‘dc:’ is the prefix label, and ‘<http://purl.org/dc/elements/1.1/>’ is an IRI reference. ‘BASE’ is the keyword for defining a base IRI. Once the SparQL Parser detects a definition of a prefix label, it creates a data structure to save the mapping relationship between the pair of the prefix label and the IRI reference. This created data structure is attached to a list that keeps all of the pairs of prefix labels to IRI references in one SPARQL statement. After that, when the SparQL Parser finds

any prefixed names in the statement, it replaces the prefix label by its corresponding IRI reference. Since there is at most one definition of a base IRI in one SPARQL statement, the SparQL Parser saves the base IRI the first time it is encountered and reports an error if base IRI is found more than once in a SPARQL statement. For example,

```
BASE <http://example.org/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
<department>
dc:people
```

‘dc:people’ is interpreted as <http://purl.org/dc/elements/1.1/people> and the relative IRI <department> is interpreted as <http://example.org/department>. In the SparQL Parser, both prefixed names and relative IRIs are translated to IRIs.

5.2. Variable Collector

Variables in SPARQL queries have global scope. Variables are indicated by "?"; the "?" does not form part of the variable. "$" is an alternative to "?". In a query, $abc and ?abc are both variable named ‘abc’.

There is no particular syntax in a SPARQL statement to define the data type of variables, which means it is impossible to know which data type a variable is going to have. By contrast, all variables used in an AmosQL statement must be declared in variable declaration commalist [4]. In the SparQL Parser, every variable name that appeared in a SPARQL query is registered in the variable collector. The SPARQL variable data type is unknown until it is bound to a value; it can be URI reference saved in Amos II type ‘Resource’, blank node saved in Amos II type ‘Resource’, or literal saved in Amos II type ‘Rlit’. Therefore all variables are declared as Amos II type ‘Resource’, the super class of both ‘Resource’ and ‘Rlit’ in the Amos II. For example, SPARQL variables

```
?name, ?email, ?address
```

are declared in the from clause in the geneated AmosQL statement as:

```
…
from Resource name, Resource email, Resource address
…
```

5.3. Triple Collector and Triple Pattern

The building blocks of SPARQL queries are triple patterns. The following is a basic triple pattern having a subject variable, a predicate URI reference and an object variable:

```
?book <http://example.com/title> ?title.
```

A period '.' denotes the end of the expression of one or several triple patterns.

This is translated to the AmosQL 3-tuple expression:

```
<book, uri('http://example.com/title'), title>
```

Furthermore, there are two syntactic forms that abbreviate sequences of query triples, namely *predicate-object list* and *object list*. Predicate-object lists denote situations when triple patterns share a common subject so that a subject is written once and used for more than one triple pattern by employing the ';' notation. For example,

```
?book <http://example.com/title> ?title;
      <http://example.com/price> ?price.
```

This is the same as writing the triple patterns:

```
?book <http://example.com/title> ?title.
?book <http://example.com/price> ?price.
```

Therefore, they should be translated to two AmosQL 3-tuples:

```
<book, uri('http://example.com/title'), title>
<book, uri('http://example.com/price'), price>
```

If triple patterns share both subject and predicate these can be expressed in the object list format using the ',' notation.

```
?book  <http://example.com/title>  'Records  of  Three  Kingdoms',
                                   'Journey to the West'.
```

is the same as

```
?book <http://example.com/title> 'Records of Three Kingdoms'.
?book <http://example.com/title> 'Journey to the West'.
```

A more complex situation is that an object list is combined with predicate-object lists:

```
?book <http://example.com/price> ?price;
      <http://example.com/title>  'Records  of  Three  Kingdoms',
                                  'Journey to the West'.
```

This forms three triple patterns:

```
?book <http://example.com/price> ?price.
?book <http://example.com/title> 'Records of Three Kingdoms'.
?book <http://example.com/title> 'Journey to the West'.
```

Their corresponding AmosQL 3-tuples are:

```
<book, uri('http://example.com/price'), price>
<book, uri('http://example.com/title'),
                                  r(1,'Records of Three Kingdoms')>
<book, uri('http://example.com/title'), r(1,'Journey to the West')>
```

During parsing a SPARQL statement, the SparQL Parser cannot predicate the format of the triple pattern until the end of statement marker period is found. The triple patterns can be expressed in the basic triple pattern, in the predicate-object list, or in the object list. To collect the information of the triple patterns accurately, SparQL Parser utilizes the idea of *triple pattern tree*.

A triple pattern tree is a tree data structure whose height is two. Its root node is a subject, nodes in level one of the tree are the predicates, and the nodes in level two are the objects. When the SparQL Parser finds triple patterns, it maps the subject, predicates, and objects to a triple pattern tree structure. The object, the predicates, and the objects of the triple patterns in the previous example are therefore saved in the triple pattern tree structure in Figure 12.

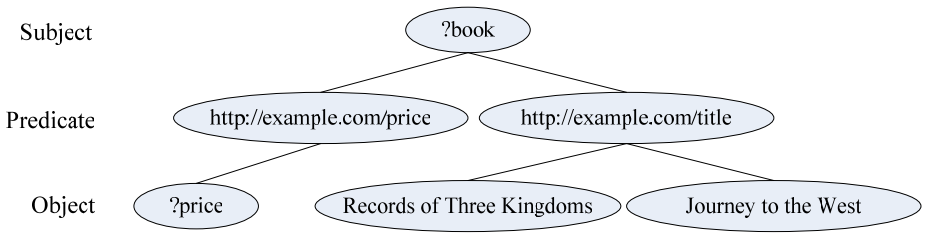


Figure 11 Example of Triple Pattern Tree

Once the SparQL Parser reaches the end of statement marker, it progresses a post-order walk on the triple pattern tree. One triple pattern is generated when the tree reaches a leaf node, the RDF triple object. Its parent node is the predicate. The root node is the subject. Figure 13 shows three traverse routes ‘1’, ‘2’, and ‘3’, which generates three tuples.

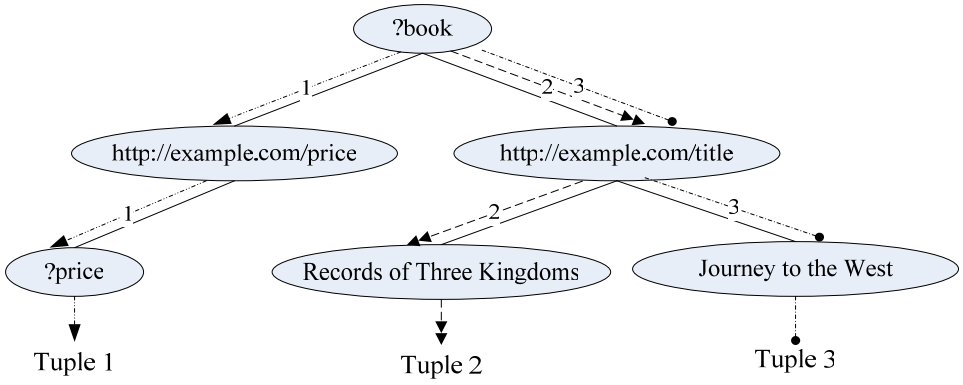


Figure 12 Example of Triple Pattern Tree

5.4. Basic Graph Patterns

A *basic graph pattern* is a set of triple patterns. It forms the basis of SPARQL query matching, which is to find a mapping from blank nodes and variables in the basic graph pattern to terms in the RDF source, the graph being matched. A simple SPARQL query statement contains a basic graph pattern and a *query-variable list* of names of variables to be queried. Values

bound to the variables in the query-variable list forms a set of results to the SPARQL query. The basic steps that the SparQL Parser takes to convert a SPARQL query statement to AmosQL statement include:

- Collecting all of the variables that appeared in the SPARQL query and declare them as Amos II type ‘Resource’ in the ‘from_clause’ of AmosQL
- Converting the query-variable list to 'variablecomma_list' in AmosQL
- Converting triple patterns to predicates in the AmosQL 'where_clause'

For example, suppose there is a SPARQL query over RDF document ‘c:/test.rdf’:

```
SPARQL query 2
    SELECT  ?title
    WHERE
    {<http://example.org/book/book1>
         <http://purl.org/dc/elements/1.1/title>
           ?title. }
```

Its corresponding AmosQL statement is:

```
AmosQL query 3

    select{title}①
    from Resource title
    where rdf_tr②("c:/test.rdf")=
        <uri("http://example.org/book/book1"),
        uri("http://purl.org/dc/elements/1.1/title"),
        title>;
```

IRIs have two abbreviated formats (explained in section 2.1.3 and section 5.1) that shorts SPARQL query statement. The following example shows how SPARQL query 2 is structured if prefixed names or relative names are used, respectively.

```
SPARQL query 3
    PREFIX ebook: <http://example.org/book/>
    PREFIX purl: <http://purl.org/dc/elements/1.1/>
    SELECT  ?title
    WHERE
        { ebook:book1 purl:title ?title. }

SPARQL query 4
    BASE <http://example.org/book/>
    PREFIX purl: <http://purl.org/dc/elements/1.1/>
    SELECT  ?title
    WHERE{ <book1> purl:title ?title. }
```

① Why query results are returned in a AMOS II type ‘vector’ will be explain in section 5.8.
② Function ‘rdf_tr’, using function ‘parseRDF’ in CRDF, converts the tiples in one RDF source to 3-tuples and emits these 3-tuples. For more details, please read section 5.10.

That the SPARQL statement is expressed in either abbreviated formats does not change their corresponding AmosQL. Therefore, their corresponding AmosQL statement is exactly like AmosQL query 3.

The next example is a more complex SPARQL query statement, which contain two triple patterns.

```
SPARQL query 5
    PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
    SELECT  ?name ?mbox
    WHERE
      { ?x foaf:name ?name;
           foaf:mbox ?mbox .
      }
```

This is translated to

```
AmosQL query 4
    select{name,mbox}
    from Resource name, Resource mbox, Resource x
    where
    rdf_tr("c:/test.rdf")=
        <x,uri("http://xmlns.com/foaf/0.1/name"),name> and
    rdf_tr("c:/test.rdf")=
    <x,uri("http://xmlns.com/foaf/0.1/mbox"),mbox>;
```

5.5. Literals in a SPARQL query

In SPARQL statements, literals are presented in the same way as they are in RDF source as follows:

- plain literal: ‘abc’
- plain literal with language tag: ‘abc’@en-gb
- xsd:integer: ‘10’^^xsd:integer
- xsd:decimal: ‘1.1’^^xsd:decimal
- xsd:double: ‘1.1e0’^^xsd:double
- xsd:boolean: ‘true’^^xsd:boolean
- xsd:string: ‘happy’^^xsd:string

The SparQL Parser defines an AmosQL function, ‘newLit(Charstring)->Rlit’ to create Amos II type ‘Resource’ from Amos II type ‘Charstring’, which includes both the value string and the data type URI. For example,

newLit(“10’^^xsd:integer’) creates *r(2, ‘10’)*
newLit(“1.1’^^xsd:decimal’) creates *r(3, ‘1.1’)*
newLit(“1.1e0’^^xsd:double’) creates *r(4, ‘1.1e0’)*

Literals of data type xsd:integer, xsd:decimal, or xsd:double expressed in the abbreviated format will be revert to their normal expression, before ‘newLit’ is called. For example,

10 is reverted to ‘10’^^xsd:integer then *newLit(“10’^^xsd:integer’)* is called
1.1 is reverted to ‘1.1’^^xsd:decimal then *newLit(“1.1’^^xsd:decimal’)* is called
1.1e0 is reverted to ‘1.1e0’^^xsd:double then *newLit(“1.1e0’^^xsd:double’)* is called

For xsd:boolean, *‘true’^^xsd:boolean* and *‘1’^^xsd:boolean* are considered equal, so are *‘false’^^xsd:boolean* and *‘0’^^xsd:boolean* in a SPARQL query statement.

A SPARQL query may ask for the value string of data type URI of a literal. Therefore, the SparQL parser provides two functions that accept a ‘Resource’ to retrieve its value string or data type URI. They are defined as:

```
str(Resource x)->Charstring
dt(Resource res)->Integer
```

For example, to retrieve the value string and data type URI of r(2, ‘10’), we have:

```
str(r(2, ‘10’)) returns ‘10’
dt(r(2, ‘10’)) returns 2
```

Now SparQL Parser knows how to convert an RDF literal in a SPARQL query to AmosQL. However, representing typed literal as Amos II type ‘Rlit’ brings a difficulty to the implementation of the SparQL Parser. Suppose there is an RDF resource ‘c:/test.rdf’, which has two triples, stored in the database:

```
<uri(‘aaa’), uri(‘bbb’), r(4,‘‘10.0’^^xsd:double’)>
<uri(‘ccc’), uri(‘bbb’), r(4,‘‘0.10e2’^^xsd:double’)>
```

Then, there is a SPARQL query over the RDF resource:

```
SPARQL query 6
    SELECT ?x
    WHERE {?x <bbb> 1.0e1.}
```

whose corresponding AmosQL query is:

```
AmosQL query 5
    select{x}
    from Resource x
    where
      rdf_tr("c:/test.rdf")=<x,uri(“bbb”),r(4,‘‘1.0e1’^^xsd:double’)>;
```

Should AmosQL query 5 find the correct variable bindings (x = uri(‘aaa’) and x = uri(‘bbb’)), according to the literal matching mentioned in section 2.4? The answer is NO, because typed literals are stored as Amos II type ‘Rlit’ for which it holds that,

```
r(4,‘‘1.0e1’^^xsd:double’) =/= r(4,‘‘10.0’^^xsd:double’)
r(4,‘‘1.0e1’^^xsd:double’) =/= r(4,‘‘0.10e2’^^xsd:double’)
```

Whereas, if the object is unknown therefore is a variable in the SPARQL statement as,

```
SPARQL query 7
    SELECT ?x,?num
    WHERE {?x <bbb> ?num.}
```

which is translated to AmosQL query:

```
AmosQL query 6
    select{x,num}
    from Resource x, Resource num
    where rdf_tr("c:/test.rdf")=<x,uri(“bbb”), num>;
```

This time, AmosQL finds the correct variable bindings.

```
    { uri(‘aaa’), r(4,‘‘10.0’^^xsd:double’)}
    { uri(‘bbb’), r(4,‘‘0.10e2’^^xsd:double’)}
```

To solve this problem, in addition to the functions ‘rdf_tr’ and ‘rdf_tr_c’, the SparQL Parser introduces two more derived functions to deal with triple patterns when objects are literals. Literal objects become input parameters that restrict the value binding of subject or predicate. These functions are shown in Table 2.

Table 2 Amos II Functions in Pattern Matching

Function prototype	Subject	Predicate	Object
rdf_tr(Charstring src, Resource o)-><Resource s, Resource p>	—	—	literal
rdf_tr(Charstring src)-><Resource s, Resource p, Resource o >	—	—	others
rdf_tr_c(Charstring src, Resource o)-><Resource s, Resource p>	—	—	literal
rdf_tr_c(Charstring src)-><Resource s, Resource p, Resource o >	—	—	others

The first ‘rdf_tr’ function in Table 2 is defined as:

```
    create function rdf_tr(Charstring src, Resource o)
        -> <Resource,Resource> as
        select sub, pre from Resource sub, Resource pre, Resource obj
        where parseRDF(src)=<sub,pre,obj> and
              fltr(EQ(obj,o))① and _dt_(obj)=_dt_(o);
```

The function *fltr* determines the logical value of an expression, the function *EQ* compares two RDF resources and the function *_dt_* returns the RDF type tag of a resource.

① AmosQL function ‘fltr’ and ‘EQ’ will be introduced in section 5.6

The SPARQL query 6 will be converted to AmosQL as,

```
select{x}
from Resource x
where
  rdf_tr("c:/test.rdf",newLit(‘‘1.0e1’^^xsd:double’))=<x,uri(“bbb”
  )>;
```

The function ‘rdf_tr’ makes sure that two typed literals are of the same data type and are equal.

5.6. Filter Collector and Value Constraints

Basic graph pattern matching creates bindings of variables. It is possible to restrict solutions by constraining the allowable bindings of variables to RDF terms. Value constraints, which are enclosed by ‘(’ and ‘)’ after keyword ‘FILTER’, take the form of Boolean-valued expressions; The SparQL Parser also allows user-defined functions that return typed literal xsd:boolen to form the value constraint. For example, here is a SPARQL query over ‘c:/test.rdf’ that asks for the title and price of books whose price is less than 30,

```
SPARQL query 8
   PREFIX  dc: <http://purl.org/dc/elements/1.1/>
   PREFIX  ns: <http://example.org/ns#>
   SELECT  ?title ?price
   WHERE
       { ?x ns:price ?price .
         FILTER ( ?price < 14+16 ) .
         ?x dc:title ?title .
       }
```

There is a set of functions and operations in the SparQL Parser for value constraints, however currently the SparQL Parser does not support all kinds of constraints. All supported contraints are introduced respectively in section 5.6.2, 5.6.3, 5.6.4 and 5.6.5. In the SparQL Parser, it is the filter collector module that stores expressions of value constraints in a data structure called *Filter*. Each filter corresponds to one ‘FILTER’ keyword. Then the SparQL Parser translates SPARQL operators and functions to their corresponding AmosQL functions if there is a mapping, otherwise, the SparQL Parser reports an error.

The translation of filters will be explained next.

5.6.1. Effective Boolean Value

SPARQL functions requiring an argument of type xsd:boolean, such as ‘!’ or logical a operator, coerce the input argument to xsd:Boolean, i.e. any value can be regarded as a Boolean value in SparQL An *effective boolean value* (EBV) of an expression determines its truth value. The EBV rules are described in section 11.2.2 in [2]. The SparQL Parser has an

AmosQL function named ‘EBV’ that accepts ‘Rlit’ as its input argument and returns an ‘Rlit’ indicating the result. The function ‘ebv’ follows the rules:

- If the argument is a typed literal with a data type of xsd:boolean, the EBV is the argument.
- If the argument is a plain literal or a typed literal with a data type of xsd:string, the EBV is false if the operand value has zero length; otherwise, the EBV is true.
- If the argument is a numeric type or a typed literal with a data type derived from a numeric type, the EBV is false if the operand value is numerically equal to zero; otherwise, the EBV is true.
- All other arguments produce a false.

For example, the following function calls return ‘Rlit’ r(5, ‘true’),

```
EBV(r(1,"abc"));
EBV(r(5,"true"));
EBV(r(6,"abc"));
EBV(r(4,"1.23"));
EBV(r(2,"123"));
```

By contrast, the following calls return ‘Rlit’ r(5, ‘false’)

```
EBV(r(1,""));
EBV(r(5,"false"));
EBV(r(6,""));
EBV(r(4,"0.0"));
EBV(r(2,"0"));
```

The EBV is used to calculate the arguments to SPARQL logical expressions. The SparQL Parser has a special function to evaluate the result of filters in a query. This function is named as ‘fltr’, which works just as function ‘EBV’, but, instead of returning Amos II type ‘Rlit’ in ‘EBV’, ‘fltr’ returns Amos II type ‘boolean’.

Thus, SparQL Parser translates the SPARQL query 8 to AmosQL query 7 as:

```
AmosQL query 7
    select{title,price}
    from Object title,Object price,Object x
    where
    rdf_tr("c:/test.rdf")=
      <x,uri("http://example.org/ns#price"),price> and
    rdf_tr("c:/test.rdf")=
      <x,uri("http://purl.org/dc/elements/1.1/title"),title> and
    FLTR(LT(price,
         newLit(‘‘14’^^xsd:integer’)+newLit(‘‘16’^xsd:integer’)
        ));
```

5.6.2. Logical Connectives

There are two logical connectives in SPARQL queries, && and ||, as shown in Table 3. The AmosQL function ‘l_o’ performs an ‘or’ operation on two xsd:boolean literals, while the function ‘l_a’ performs an ‘and’ operation. Inasmuch ‘l_o’ and ‘l_a’ accepts xsd:boolean literal, any other typed literal as input will be coerced to xsd:boolean following the EBV rules.

Table 3 Supported Logical Connectives

SPARQL Function	**SPARQL Type of A**	**SPARQL Type of B**	**Amos II Function**	**AmosQL Result type**
Logical Connectives				
A || B	xsd:Boolean	xsd:Boolean	l_o(A, B)	Rlit
A && B	xsd:Boolean	xsd:Boolean	l_a(A, B)	Rlit

For example, we ask for the result of operation ‘or’ and operation ‘and’ on plain literal ‘0’ and xsd:integer 0:

```
l_o(r(1,‘0’),r(2,‘0’)); returns r(5, ‘true’)
l_a(r(1,‘0’),r(2,‘0’)); returns r(5, ‘false’)
```

5.6.3. Comparison Tests

The SPARQL query language supports comparison tests on two literals with the same data type URI. The SparQL Parser defines its logical operators corresponding to those defined in the SPARQL query language as shown in Table 4. All of these functions accept two Amos II objects of type ‘Rlit’ as the input arguments and return ‘true’^^xsd:boolean or ‘false’^^xsd:boolean as Amos II type ‘Rlit’. URI references are compared by their resource identification; plain literals and xsd:strings are compared by their value string; numeric[ⓘ]values are compared by their values.

Table 4 Supported Comparison Tests

SPARQL Function	**SPARQL Type of A**	**SPARQL Type of B**	**Amos II Function**	**AmosQL Result type**
Comparison Tests				
A = B	URI reference Literal	URI reference Literal	EQ(A, B)	Rlit
A != B	URI reference Literal	URI reference Literal	NEQ(A, B)	Rlit
A < B	Plain Literal Numeric xsd:string	Plain Literal Numeric xsd:string	LT(A, B)	Rlit

[ⓘ] Numeric denotes typed literal with data types xsd:integer, xsd:decimal, and xsd:double

SPARQL Function	SPARQL Type of A	SPARQL Type of B	Amos II Function	AmosQL Result type
A > B	Plain Literal Numeric xsd:string	Plain Literal Numeric xsd:string	GT(A, B)	Rlit
A <= B	Plain Literal Numeric xsd:string	Plain Literal Numeric xsd:string	LTE(A, B)	Rlit
A >= B	Plain Literal Numeric xsd:string	Plain Literal Numeric xsd:string	GTE(A, B)	Rlit

Here are some example of comparison tests:

```
EQ(r(5, 'false'),r(5, 'false')); returns r(5, 'true')
LT(newLit('ba'),newLit('b')); returns r(5, 'false')
GTE(r(2,'10'),r(4,'10.0e0')); returns r(5, 'true')
```

```
SPARQL query 9
    PREFIX  dc: <http://purl.org/dc/elements/1.1/>
    PREFIX  ns: <http://example.org/ns#>
    SELECT  ?title ?price
    WHERE
      { ?x dc:title ?title;
           ns:discount ?discount;
           ns:price ?price.
        FILTER ( ?price < 300 && ?discount < 0.9) .
     }
```

Is interpreted as AmosQL

```
AmosQL query 8
    select{title,price}
    from Resource title,Resource price,Resource x,Resource discount
    where rdf_tr("c:/test")=
          <x,uri("http://purl.org/dc/elements/1.1/title"),title>and
       rdf_tr("c:/test")=
          <x,uri("http://example.org/ns#discount"),discount>and
       rdf_tr("c:/test")=
          <x,uri("http://example.org/ns#price"),price>and
       FLTR((l_a(LT(price,r(2,"300")),LT(discount,r(3,"0.9"))))));
```

5.6.4. Arithmetic Operators

The SPARQL query language supports arithmetic operators on numeric typed literals, namely, xsd:integer, xsd:decimal, xsd:double, as shown in Table 5.

Table 5 Supported Arithmetic Operators

SPARQL Function	SPARQL Type of A	SPARQL Type of B	Amos II Function	AmosQL Result type
Arithmetic Operators				
A * B	Numeric	Numeric	times(A , B)	Rlit
A / B	Numeric	Numeric	div(A , B)	Rlit
A + B	Numeric	Numeric	plus(A , B)	Rlit
A - B	Numeric	Numeric	minus(A , B)	Rlit

SparQL Parser overloads Amos II built-in functions ‘times’, ‘div’, ‘plus’, and ‘minus’ so that they accept Amos II type ‘Rlit’ as their input arguments and return Amos II type ‘Rlit’ of proper typed literal. A proper typed literal result is defined as:

- Times operation: when both of the two input arguments are of type xsd:integer, the return type is xsd:integer; Otherwise, the return type is xsd:double.
- Div operation: when both of the two input arguments are of type xsd:integer the return type is xsd:decimal; Otherwise, the return type is xsd:double.
- Plus operation: when one of the two input arguments is of type xsd:double, the return type is xsd:double; when one of the two input arguments is of type xsd:decimal, the return type is xsd:decimal; Otherwise, the return type is xsd:integer.
- Minus operation: the same as the rule for plus.

For example,

```
r(3,‘100’)+r(2,‘100’); returns r(3,‘200’)
r(4,‘10’)*r(2,‘10’); returns r(4,‘100.0’)
r(2,‘10’)/r(2,‘10’); returns r(3,‘1.000’)
```

```
SPARQL query 10
    PREFIX  dc: <http://purl.org/dc/elements/1.1/>
    PREFIX  ns: <http://example.org/ns#>
    SELECT  ?title ?price
    WHERE
      { ?x dc:title ?title;
           ns:discount ?discount;
           ns:price ?price.
        FILTER ( ?price * ?discount < 300) .
```

Is interpreted as

```
AmosQL query 9
    select{title,price}
    from Resource title,Resource price,Resource x,Resource discount
    where
    rdf_tr("c:/test")=
       <x,uri("http://purl.org/dc/elements/1.1/title"),title>and
    rdf_tr("c:/test")=
```

```
   <x,uri("http://example.org/ns#discount"),discount>and
rdf_tr("c:/test")=
   <x,uri("http://example.org/ns#price"),price>and
FLTR((LT(price*discount,r(2,"300"))));
```

5.6.5. Other SPARQL tests

Table 6 shows other supported SPARQL functions and their corresponding AmosQL functions. The functions follow the rules defined in [2]. There can the readers find more details on the functionalities and constrains on all functions.

Table 6 Other Supported Functions

SPARQL Function	SPARQL Type of A	Amos II Function	Result type
! A	xsd:boolean	fn_not(A)	Rlit
isIRI(A)	RDF term	isIRI(A)	Rlit
isURI(A)			
isBLANK(A)	RDF term	isBlnk(A)	Rlit
isLITERAL(A)	RDF term	isLtrl(A)	Rlit
STR(A)	literal	STR(A)	Rlit
	IRI		
LANG(A)	literal	LANG(A)	Rlit
DATATYPE(A)	typed literal	DT(A)	Rlit
langMATCHES(A, B)	simple literal	simple literal	Rlit

For example,

```
fn_not(r(3,'1.0')); returns r(5, ‘true’)
isIRI(uri('http://a.com')); returns r(5, ‘true’)
isBLNK(uri('http://a.com')); returns r(5, ‘false’)
str(r(2,‘01’)); returns r(1,‘01’)
lang(r(1,‘‘abc’@en-gb’)) returns r(1,‘en-gb’)
```

Here we give an example of a SPARQL statement querying for the title and price of a book written in English:

```
SPARQL query 11
   PREFIX  dc: <http://purl.org/dc/elements/1.1/>
   PREFIX  ns: <http://example.org/ns#>
   SELECT  ?title ?price
   WHERE
   { ?x dc:title ?title;
        ns:price ?price.
     FILTER ( ?price * 0.8 < 300 && langMATCHES(Lang(?title), 'en')) .
   }
```

Is interpreted as

```
AmosQL query 10
    select{title,price}
    from Resource title,Resource price,Resource x
    where
        rdf_tr("c:/test")=
            <x,uri("http://purl.org/dc/elements/1.1/title"),title>and
        rdf_tr("c:/test")=
            <x,uri("http://example.org/ns#price"),price>and
        FLTR((langMatches(LANG(title),r(1,'en'))));
```

5.7. Optional Graph Patterns

In the SPARQL syntax, basic graph patterns allow making queries where entire pattern must match. When the SparQL Parser finds the reult to a query, every variable must be bound to an RDF term, and later the value constraints, if any, must be satisfied. Whereas, for Web resources, complete structures cannot be assumed in all cases. It is reasonable to allow adding information to the solutions where information is available, but not to fail to find any solution because of some part of the graph pattern is not matched. *Optional graph patterns* provides this facility. If the optional part of a SPARQL query does not lead to any solutions, variables can be left unbound [2], which does not have the solution rejected unless variables in basic graph pattern is unbound or value constraints test fails.

Optional parts of the graph pattern may be specified with the keyword ‘OPTIONAL’ applied to a graph pattern:

```
SPARQL query 12
    PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
    SELECT  ?name ?mbox
    WHERE
    { ?x foaf:name ?name.
      OPTIONAL { ?x foaf:mbox ?mbox. }
    }
```

SPARQL query 12 can find results, in which the variable ‘mbox’ is unbound.
Furthermore, there can be more than one optional graph pattern in a SPARQL query statement and an optional graph pattern can include a value constraint. For example,

```
SPARQL query 13
    PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
    SELECT  ?name ?age ?mbox
    WHERE
    { ?x foaf:name ?name.
      FILTER ( langMATCHES(LANG(?name), 'en') )
      OPTIONAL { ?x foaf:mbox ?mbox }
```

```
    OPTIONAL { ?x foaf:age ?age. FILTER ( ?age >= 18) }
  }
```

For SPARQL query 13, a solution result can be generated even if variable ‘?age’ or ‘?mbox’ are not bound.

For SPARQL queries that have optional graph patterns, the SparQL Parser introduces the idea of a *Grouped Graph Pattern* and a *Grouped Value Constraint*. In section 5.3 and section 5.4, we mentioned that a basic graph pattern is a set of triple patterns and how triple patterns are described in the triple pattern tree structure. Section 5.6 describes how the SparQL Parser translates value constraints. Now, we tag the triple patterns and the filters (see section 5.6) with a group number to separate the basic graph pattern from the optional graph patterns. The SparQL Parser tags the triple patterns in basic graph pattern and the filters outside the optional graph pattern with group number ‘1’; tags triple patterns and filters in optional graph patterns consecutively from number ‘2’. For SPARQL query 13, there are three triple patterns, and two filters. Table 7 shows how they are tagged.

Table 7 Grouped Triple Patterns and Filter Sets

Triple Patterns	
1	`?x foaf:name ?name`
2	`?x foaf:mbox ?mbox`
3	`?x foaf:age ?age`
Filter Sets	
1	`langMATCHES(LANG(?name), 'en')`
3	`?age >= 18`

Figure 13 demonstrates an SPARQL query that has six triple patterns tagged number ‘1’ in basic group pattern, one triple pattern tagged number ‘2’ in optional graph pattern and two triple patterns in another optional graph pattern.

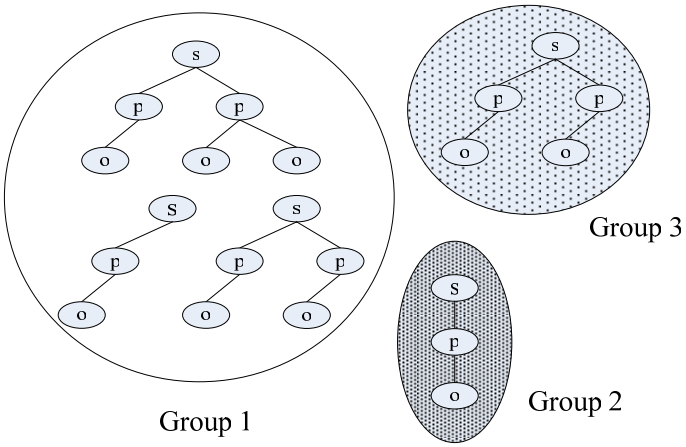


Figure 13 Example of Tagged Triple Pattern Tree

Currently, triple patterns and value constraints in optional group patterns are collected and are translated to where clause in the AmosQL query statements. However, all variables, including

those in optional graph patterns, must be bound in the solution. For example,

```
PREFIX  foaf:  <http://xmlns.com/foaf/0.1/>
SELECT ?mbox ?name
   {
     ?x foaf:mbox ?mbox .
     OPTIONAL { ?x foaf:name  ?name } .
   }
```

is interpreted as

```
select{mbox,name}
from Resource mbox,Resource name,Resource x
where
rdf_tr_c("c:/test.rdf")=
    <x,uri("http://xmlns.com/foaf/0.1/mbox"),mbox> and
rdf_tr_c("c:/test.rdf")=
    <x,uri("http://xmlns.com/foaf/0.1/name"),name>;
```

5.8. Solution Modifier

Up till now, the SparQL Parser can translate a SPARQL query that contains basic graph patterns and value constrains to a corresponding AmosQL query properly. It also correctly interprets plain literals and typed literals. We can have a set of results by executing the AmosQL query that the SPARQL query is translated to. There are sometimes needs to sort the query results on the values of one or several variables. For instance, a SPARQL query can ask for the names of the students, whose math exam results are from 10^{th} to 20^{th} in a class. The keywords ‘ORDER BY’, ‘LIMIT’, and ‘OFFSET’(see section 2.3) in SPARQL helps to achieve this requirement. The query statement can be like:

```
SPARQL query 14
    BASE <http://example.edu/>
    PREFIX foaf: <http://xmlns.com/foaf/0.1/>
    SELECT ?name ?rslt
    WHERE { ?x foaf:name ?name;
              <math> ?rslt.
          }
    ORDER BY ?rslt
    LIMIT 10
    OFFSET 10
```

The keywords ‘LIMIT’ and ‘OFFSET’ are not supported yet. However, the SparQL Parser utilizes Amos II built-in function ‘sortbagby’ to implement the sorting ability that SPARQL ‘ORDER BY’ offers.

Function prototype of ‘sortbagby’ is:

```
sortbagby(Bag b, Integer pos, Charstring order) -> Vector
sortbagby(Bag b, Vector of Integer pos, Vector of Charstring order)
                                                    -> Vector
```

Argument ‘b’ is a bag of results to be sorted; argument ‘pos’ indicates the position number in a result tuple of the bag ‘b’; argument ‘order’ indicates the ordering direction. For example,

```
sortbagby(
(select i,j
from Number i, Number j
where (i=1 or i=2 or i=3) and (j=1 or j=2)),
1,'dec');
```

returns {{3,2},{3,1},{2,2},{2,1},{1,2},{1,1}}

```
sortbagby(
(select i,j
from Number i, Number j
where (i=1 or i=2 or i=3) and (j=1 or j=2)),
{2,1},{'inc','dec'});
```

returns {{3,1},{2,1},{1,1},{3,2},{2,2},{1,2}}.

With the help of Amos II function ‘sortbagby’, it is very easy to interpret SPARQL keyword ‘ORDER BY’. For example,

```
SPARQL query 15
    BASE <http://example.org/>
    PREFIX foaf: <http://xmlns.com/foaf/0.1/>
    SELECT ?title ?price ?discount
    WHERE { ?x foaf:title ?title;
              <price> ?price;
              <discount> ?discount.
          }
    ORDER BY DESC(?discount) ASC(?price)
```

is translated to AmosQL:

```
AmosQL query 11
    sortbagby(
    (select{title,price,discount}
    from Resource title,Resource price,Resource discount,Resource x
    where
    rdf_tr("")=<x,uri("http://xmlns.com/foaf/0.1/title"),title>and
    rdf_tr("")=<x,uri("http://example.org/price"),price>and
    rdf_tr("")=<x,uri("http://example.org/discount"),discount>),
    {3,2},{"dec","inc"});
```

For the reason that function ‘sortbagby’ accepts vectors only, the SparQL Parser translates SPARQL query statement to AmosQL query statement that returns the result tuples As vectors, which is done by delimit the variablecomma_list by ‘{‘ and ’}’.

5.9. Memory Manager

The memory manager allocates memory while parsing one SPARQL statement to an AmosQL statement. Instead of using ‘malloc’, any allocated memory block operation in the SparQL Parser invokes ‘sparql_new’, an interface of the memory manager. The benefit of using ‘sparql_new’ is that no explicit free memory operation is needed. Allocated memory blocks are freed every time when the memory manager is initialized before parsing a new SPARQL query and at the end of the execution of the SparQL Parser. As a result, no memory leak will occur. Maximum size of allocated memory is decided by macro ‘MEMORY_SIZE’, which is 4K.

5.10. Usage of the SparQL Parser and Result Format

The SparQL Parser provides two functions to execute SPARQL query statements. The one that accepts SPARQL query statement as input argument is

```
sparql(Charstring rdf,Integer bCache,Charstring query)->Vector of Object
```

A SPARQL query statement is entered as a Amos II type ‘Charstring’ stored in argument ‘query’; the argument ‘rdf’ refers to the RDF source file name; the ‘bCache’ indicates whether RDF source is cached. For example,

```
sparql("c:/test.rdf", 1,
"
PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
SELECT  ?name
WHERE
{
  ?x foaf:name ?name.
  FILTER ( langMATCHES(LANG(?name), \"en\") )
}
");
```

Similarly, there is another SparQL Parser function that takes SPARQL query statement stored in a file as input argument:

```
sparql_from_file(Charstring rdf,Integer bCache,Charstring fn)->
                        Vector of Object
```

Argument ‘fn’ refers to the file name; argument ‘rdf’ refers to the RDF source file name; argument ‘bCache’ indicates whether the RDF source is cached. For example,

```
sparql_from_file(‘c:/test.rdf’, 0, ‘c:/sparql_query.rq’);
```

Inside these two Amos II functions, the external functions ‘__sparql__’ and ‘__sparql_from_file__’ are invoked to translate the SPARQL query statement to an AmosQL query statement. The AmosQL query statement is returned as an Amos II type ‘Charstring’ and is evaluated by calling Amos II built-in function ‘eval’ so as to find the query result of the SPARQL query.

The result format is a vector of objects and the query returns a bag of vectors of objects. For example, suppose RDF source file ‘c:/test.rdf’ has the following triples

```
_:P1110 <http://xmlns.com/foaf/0.1/name> "yuca"
_:P1110 <http://xmlns.com/foaf/0.1/mbox> "yuca@uu.se"
_:P1022 <http://xmlns.com/foaf/0.1/name> "yawa"
_:P1022 <http://xmlns.com/foaf/0.1/name> "yawa@uu.se"
```

A SPARQL query over RDF source file ‘c:/test.rdf’

```
sparql("c:/test.rdf",
"
SELECT ?name ?mbox
WHERE { ?x <http://xmlns.com/foaf/0.1/name> ?name ;
           <http://xmlns.com/foaf/0.1/mbox> ?mbox .
     }
");
```

finds two results,

{r(1,“yuca”), r(1,“yuca@uu.se”)}
{r(1,“yawa”), r(1,“yawa@uu.se”)}

6. Evaluation

6.1. CRDF

The performance of CRDF completely written in C/C++ and based on Raptor is evaluated and compared with an RDF wrapper completely written in Java and based on Jena [5]. We query for the number of triples are there in the file ‘gcl2.1.rdf’ that has 5416 lines and contains 4986 RDF triples.

The result is shown in Table 8. Testing environment is CPU: Intel® Celeron® M 1.7G, Main Memory: 512M, OS: Windows2000 Professional SP4. The query

Table 8 Execution Time of C RDF wrapper and JAVA RDF wrapper

Program	Result	Duration
JAVARDF	4986	0.331s
CRDF Release	4986	0.081s

Execution time of the command is cut by almost 75%.

6.2. SparQL Parser

The test suite used in testing the SparQL Parser is the existing official package of SPARQL query language test cases that were downloaded from link http://www.w3.org/2001/sw/DataAccess/tests/. Except for test cases for old syntax, the test suite includes 7 test cases that have already been approved as well as 74 test cases that have not been decided yet. Each test case in the package consists of three files: an RDF source file, a SPARQL query statement file, and a standard result file. The RDF source file and the standard result file can be in format RDF/XML, N-Triples, or Turtle. Each SPARQL query statement file is a plain text file that includes one and only one SPARQL query statement. All this information is stored in a manifest file.

The file ‘test.amosql’ under directory ‘regress’ contains AmosQL statements to parse the manifest file of the test suite, load the RDF source file, load the SPARQL query statement file, load the result file, compare the query result with the result file, and to generate a brief test report. A test with cached RDF source can be started by running batch file ‘test.bat’. Test with not cacheing RDF source can be started by running batch file ‘test_no_cache.bat’.

Test result is shown in Table 9.

Table 9 Test Result of SparQL Parser

APPROVAL	TOTAL	PASSED	PERCENTAGE
Approved	7	7	100%
Unknown	75	45	60.0%
All	82	52	63.4%

The SparQL Parser fails in 28 test cases out of a total number of 81 cases. The failure reason is shown in Figure 14. ‘UNION’, ‘REGEX’ and ‘Optional’ in legend denotes SparQL Parser not supporting keyword ‘UNION’, ‘REGEX’ and ‘OPTIONAL’ in SPARQL statement. ‘IRIrefFunc’ denotes SparQL Parser not supporting access to user-defined functions.

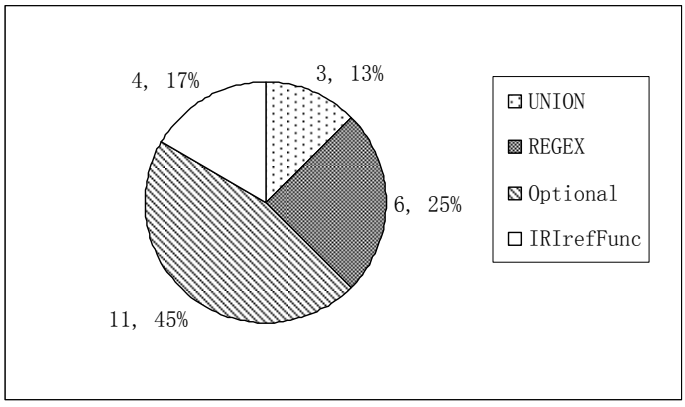


Figure 14 Failure Reason Analysis

7. Summary

We have implemented an RDF wrapper CRDF and the SparQL Parser in SPARAMOS for the Amos II mediator system. The SPARQL statements are translated into corresponding AmosQL statements calling the CRDF wrapper. The aim of CRDF is to substantially improve performance compared with JAVA version RDF parser is achieved by using the all C Raptor toolkit instead of Java based Jena. The SparQL Parser was verified by passed a majority of the standard SPARQL test suite. In particular all approved SPARQL statements in the test suite passed.

The implementation of CRDF and SPARAMOS enables access to semantic web resources from the Amos II mediator system and makes it possible to create views over web resource expressed in AmosQL. Mediator views can also be defined to combine data from different peers by using the mediation facilities of AmosQL. The SPARAMOS enables SPARQL query language to be used as an interface language to Amos II.

Future work could be to investigate how to fully support XML typed literals. Outer-join could also be an issue deserving consideration.

References

1 F.Manola, E.Miller, B.McBride. *RDF Primer W3C Recommendation 10 February 2004*. 2004. (http://www.w3.org/TR/2004/REC-rdf-primer-20040210/)

2 E.Prud'hommeaux, A.Seaborne. *SPARQL Query Language for RDF W3C Candidate Recommendation 6 April 2006*. 2006 (http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/)

3 D.Brickley, R.V. Guha, B,McBride. *RDF Vocabulary Description Language 1.0*: *RDF Schema W3C Recommendation 10 February 2004*. 2004 (http://www.w3.org/TR/rdf-schema/)

4 S. Flodin, M. Hansson, V. Josifovski, T. Katchaounov, T. Risch, and M. Sköld. *Amos II user's manual*. Uppsala University 2005 (http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html)

5 *Jena – A Semantic Web Framework for Java* (http://jena.sourceforge.net/index.html)

6 D.Beckett, B.McBride. *RDF/XML Syntax Specification (Revised) W3C Recommendation 10 February 2004*. 2004 (http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/)

7 G.Klyne, J. J.Carroll. *Resource Description Framework (RDF) Concepts and Abstract Syntax W3C Recommendation 10 February 2004*. 2004. (http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/)

8 G.Wiederhold. *Mediators in the Architecture of Future Information System*, *IEEE Computer*. March 1992

9 T.Risch, V.Josifovski. *Functional Approach to Data Management – Modeling, Analyzing and Integrating Heterogeneous Data*. Springer. ISBN 3-540-00375-4, 2003

10 R.Elmasri, S.B. Navathe. *Fundamentals of Database Systems*, *Fourth Edition*. ISBN 0321122267. Addison Wesley, 4 edition (July 23, 2003).

11 A.Seaborne. *RDQL - A Query Language for RDF W3C Member Submission 9 January 2004*. 2004 (http://www.w3.org/Submission/RDQL/)

12 D.Beckett. *Raptor RDF Parser Toolkit Overview*. 2006. (http://librdf.org/raptor/)

13 A.M. Kuchling. *Quotation Exchange Language (QEL)* (http://www.amk.ca/qel/)

14 A.Berglund, S.Boag, D.Chamberlin, M.F.Fernández, M.Kay, J.Robie, J.Siméon. *XML Path Language (XPath) 2.0 W3C Candidate Recommendation 8 June 2006*. 2006 (http://www.w3.org/TR/2006/CR-xpath20-20060608/)