IT 10 012

Examensarbete 15 hp
April 2010

Querying N-triples from an extensible functional DBMS

Martynas Mickevicius

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

Querying N-triples from an extensible functional DBMS

Martynas Mickevicius

The goal of this project is to provide an N-triple wrapper for the extensible DBMS Amos II. This enables search and filter through the N-Triple streams using the AmosQL query language to return relevant triples back as a result stream.
To implement this wrapper, first N-triples streams are parsed using the Bison/Flex parser generator. The parsed triples are streamed to Amos II using its foreign function interface. The query processing kernel of Amos II then enables general queries to N-triple data sources. The report includes a performance evaluation for the wrapper.

Handledare: Silvia Stefanova
Ämnesgranskare: Tore Risch
Examinator: Anders Jansson
IT 10 012
Tryckt av: Reprocentralen ITC

1. Introduction

Due to the always decreasing costs of managing distributed systems and continuously increasing speed of interconnectivity computer systems have become highly heterogeneous. However from the user point of view there is still a need to access and search different kinds of data. These problems can be solved by using a mediator, which is a system to enable combined search in different kinds of data sources.

There are very many different kinds of data models and standards for one database system to cope with. To alleviate this, the World Wide Web Consortium has introduced a new set of data representation languages called the *Semantic web* [2]. These languages are based on new standard called Resource Description Framework (RDF) that enables to link data of different kinds.

In this work the functional DBMS Amos II [1] is extended with a wrapper interface to allow database queries to streams of RDF triples represented in the N-Triples [4] format. The overall result of the implemented wrapper is a function that takes a stream as a parameter and returns a stream of objects which can be further manipulated in the system. The wrapper also includes primitives to create and access object properties.

2. Background

The following technologies are involved in this project:

- Resource Description Framework
- Database Management Systems
- Amos II
- Bison/Flex parser generator

2.1. Resource Description Framework

Resource Description Framework (RDF) is a method to model any kind of information. It is based on modeling data and relationships between data entities as graphs. RDF has quite a few serialization formats, such as XML and Notation 3 (N3) [3]. A subset of the latter, the N-Triples format [4] is the one being used in this project work. It has some advantages over the other formats, such as being human readable and having minimal overhead.

An N-Triples data file consists of a set of triples. N-Triples represent the same data relationships as RDF statements do. For example, the following three N-Triples define three RDF statements:

```
<http://www.w3.org/> <http://purl.org/creator> "Dave" .
<http://www.w3.org/> <http://purl.org/creator> "Art" .
<http://www.w3.org/> <http://purl.org/publisher> <http://www.w3.org/> .
```

These RDF statements can also be expressed using the more complex RDF-XML format [12]:

```
<rdf:RDF xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
         xmlns:dc="http://purl.org/">
  <rdf:Description rdf:about="http://www.w3.org/">
    <dc:creator>Art</dc:creator>
    <dc:creator>Dave</dc:creator>
    <dc:publisher rdf:resource="http://www.w3.org/"/>
  </rdf:Description>
</rdf:RDF>
```

The triples are defined using three kinds of resources: *URIs*, *literals* and *blank nodes*. They are represented in files using different notations.

2.1.1. URIs

A *uniform resource identifier*, *URI*, represents an identifier of globally unique resource on the web, e.g. a URL or an image. It is enclosed in less than (<) and greater than (>) signs, e.g.:

```
<http://example.org/resource1>
```

2.1.2. Literals

A *literal* is used to represent values such as numbers or dates by means of a lexical representation. They are specified enclosed in double quotes (“), optionally followed by type or language definitions, e.g.:

```
"chat1"^^<http://example.org/datatype1>
"chat2"@en-us
```

Note that a literal string is followed by a double caret sign (^^) and another URI resource representing the datatype.

2.1.3. Blank node

A *blank node* is an internal node which is not identified by a universal *URI* and is not *literal*. It starts with an underscore and a colon (_:) and then is followed by its name.

```
_:anon
```

2.1.4. N-Triples file format

The RDF statements in an N-Triples file are defined as triples of resources, for example:

```
<http://www.w3.org/> <http://purl.org/creator> "Dave" .
```

Each RDF statement consists of *subject*, *predicate,* and *object*. The *subject* is the resource that is being described. It can be a *URI* or a *blank node*. The *predicate* defines the property for the *subject*. It is always a *URI*. The *object* is referred as a property value for the subject. It can be either of three kinds of resources. This gives a total combination of six kinds of triples. In the N-triples format each triple is followed by dot (.) and an end-of-line character. Triple can be one of the following:

```
URI URI URI .

URI URI Literal .

URI URI Blank .

Blank URI URI .

Blank URI Literal .

Blank URI Node .
```

This specifies the complete grammar of N-Triples as Extended Backus-Naur Form (EBNF) Grammar [6].

This project implements a parser for RDF triples in the N-Triples format using this grammar.

2.2. Database Management Systems

A database management system (DBMS) is a set of programs which are responsible for storage, management, and retrieval of data from a database. A DBMS accepts *queries* for data from an external application program and delivers matching data back to the application. The most common query language is SQL.

A *data model* is the kind of data types used to model data in a DBMS. The most common data model is the relational model where data is represented as tables consisting of rows and attributes (columns). These tables can be connected to each other by matching attribute values.

This project enables processing of database queries of N-Triples files.

2.3. Amos II

Amos II is an extensible DBMS that uses a functional data model which model data in terms of types, functions, and objects. Amos II has a functional query language called ALisp.

Functions in Amos II define properties of objects. There are five different kinds of functions: *stored, derived, foreign, procedure* and *overloaded.* A *stored function* in Amos II correspond to a *table* in a relational database system. It provides attributes stored on objects. A *derived function* is analoguos to *view* in a relational database system. It is defined as a query over other functions. Amos II is easily extendable using *foreign functions*. Foreign functions are implemented in some external programming language e.g. C, Java, or Lisp and then registered to be used in AmosQL queries.

Functions in Amos II can be *multi-directional*, which allow them to be used as a normal function; taking parameters and returning result, or do the inverse – take a result and find parameters that correspond to the given result.

In this project the foreign function interface is used to enable AmosQL queries to N-Triple files.

2.4. Bison/Flex

To transform plain text to data structures, the parser Bison [7] and lexical analyzer Flex [8] are used in this project. Flex is a tool for generating very fast lexical analyzers which tokenize given text using regular expressions. Tokens can be used directly or given to a parser. The general parser generator Bison is used in this project to generate an N-triple parser.

Flex provides rules in a form of regular expressions and associated C code that is executed when the expression matches a parsed text string. Bison needs an annotated context-free grammar that describes what tokens can be expected from the lexical analyzer together with semantic rules expressed as regular C code executed when a rule is satisfied. Flex regular expressions and Bison rules are compiled into regular C code, which is hardly human readable but very efficient.

3. The N-Triple wrapper

In this project Amos II is extended to query N-triples data sources. External functions send N-triples data directly to Amos II one by one as stream. The received triples are then available to search or filter using the AmosQL query language.

Amos II includes an embedded Lisp system called ALisp interpreter [5]. It is a subset of CommonLisp. In this work ALisp is used for testing and verification.

The wrapper program is written in C to obtain the best possible performance. It has two main logical parts: the *parser* is responsible to extract RDF resources from streams, and *functions* are used to create and manipulate different kinds of RDF resources.

3.1. The parser

The parser is divided into three parts (*Figure 1*): *lexer, parser,* and *filter* of data emitted to the Amos II kernel.

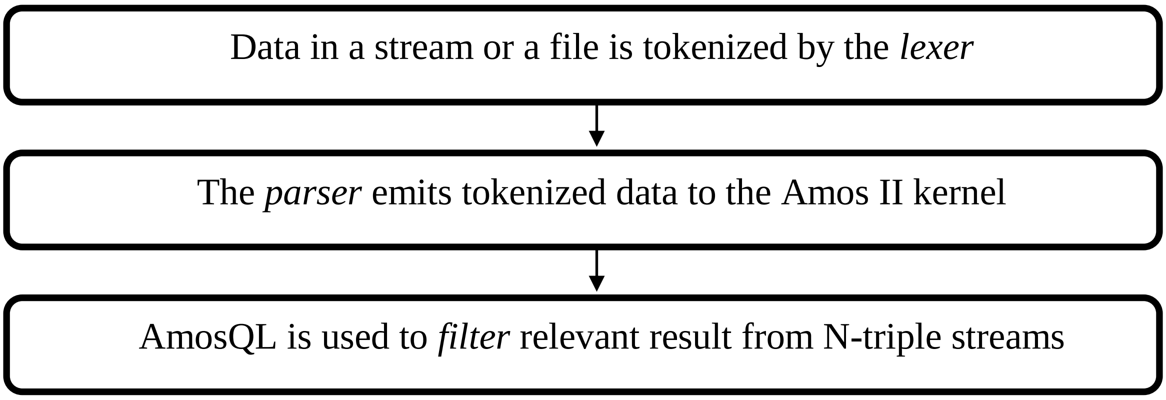
Data in a stream or a file is tokenized by the *lexer*
The *parser* emits tokenized data to the Amos II kernel
AmosQL is used to *filter* relevant result from N-triple streams

Figure 1. Architecture of wrapper

To allow the Amos II system to access and manipulate RDF data, a new data type *RDFResource* is introduced. This type represents any resource found in N-Triple streams and files. The wrapper also has functions to create, manipulate and delete objects of type *RDFResource*.

The simplest task for a wrapper is to count the number of triples in a file. This could be done by the following AmosQL query:

```
count(ntriples("nt/w3.nt"));
```

The foreign function `ntriples(Charstring name)` has only one parameter, which is a name of a file containing N-Triples data. This function converts the RDF triples in an N-Triple file to a stream of tuples containing three *RDFResource* objects.

The function first opens the specified file and executes the parser function repeatedly. The main job of a parser is to combine given tokens to more complex grammatical structures. The parser calls the *yylex()* C function to get the next tokens from the input stream. The lexer reads stream character by character trying to match any given regular expression rule. When it succeeds, it executes the C code associated with each lexer rule to identify text tokens returned to the parser.

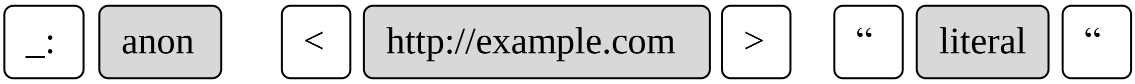

Figure 2. One line tokenized by the lexer. Grey boxes indicate tokens needed for the parser.

After every received token, the parser checks if read tokens can be combined to create a new *RDFResource* object. It does so by checking the given Bison grammar rules. When a group of three *RDFResource* objects is created a triple is emitted to the Amos II kernel in a form of a tuple of three *RDFResource* objects. The parser then frees memory and starts again to look for new tokens.

3.2. RDF types and functions

In order to provide a simple abstraction layer and to simplify manipulation of RDF resources, a type hierarchy is introduced into the wrapper program. The most important issue is that an RDF literal can have many data types (*Figure 3*) and for each data type there are a number of specific functions (operators). For example if an RDF resource represents a number basic arithmetic functions would be applicable directly on the *RDFResource* object. Other requirements occur when, e.g. a literal represents a time of day or a date.

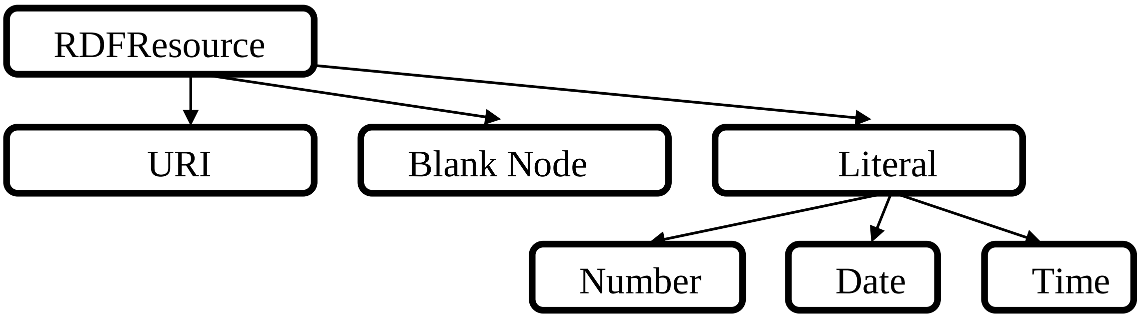

Figure 3. *RDFResource* type hierarchy.

A type hierarchy is important in this situation because the data abstraction and functions provide flexible abstractions and makes the wrapper easy to use and understand.

3.3. Encoders and decoders

It is very important to integrate the wrapper into the Amos II system as seamless as possible which means reusing Amos II kernel functions for built-in types.

An RDF *literal* can represent many kinds of objects. If a *literal* represents a number then all the arithmetic functions should be available to use with that *literal*. In order to do so, a *literal* representing a number must be encoded into a binary internal Amos II representation representing a number from the external format, which is a plain string. A function that is used to convert an external string representation of the data to an internal Amos II object is called *encoder*.

It is also possible in Amos II queries to create new RDF literals representing e.g. numbers converted into corresponing RDFResource representation. Whenever such a resource is exported to outside of the system it needs to be converted to the external string representation. A function that is used to convert an internal Amos II object to the corresponding external string representaion is called a *decoder*.

4. Implementation

4.1. RDFResource objects

The Amos II system has its own main-memory storage manager called AStorage [9] where every physical object is accessed by a *handle*. There are C functions and macros provided to manipulate these handles. A reference count garbage collector takes care of deallocating physical objects when they are no longer used. When a physical object is created, its reference counter is initialized to 0. If an object is assigned to (referenced from) some memory location its reference count is increased. If an object is no longer referenced from some location the reference counter is decreased. Only when the reference counter reaches zero, the object is physically deallocated from the database memory.

4.1.1. Internal structure

AStorage also allows creating custom types. In order to store data from N-Triples streams the new type *RDFResource* is defined. This type is the base type for all other child types representing different kinds of RDF resources. Each *RDFResource* object will represent a URI, a literal, or a blank node. A tuple of *RDFResource* objects is created from every row of an N-Triples stream.

Custom objects are represented in AStorage using C structures. There is a global *type table* managed by AStorage which saves relationships between types and their functions. In order to introduce new type, a custom C structure needs to be created to store custom data. Deallocation and printing functions for custom type also need to be provided.

The following *C* structure represents *RDFResource* objects and is called *rdfrcell*. The structure fields are used by both the system and users to store *RDFResource* data.

```
struct rdfrcell      // Template for storage type RDFRESOURCE
{
  objtags tags;      // System tags
  int kind;          // URI reference:     0
                     // literal:           1
                     // blank node:        2
                     // larger values mean different
                     // objects expressed as literals
  oidtype datatype;  // datatype for typed literals
  oidtype decoded;   // data in external format (string)
  oidtype encoded;   // data in internal format (object)
  char lang[1];       // language, if any
};
```

The first field is used by the AStorage manager and is never available for a user.

The second field *kind* stores an integer which determines the kind of the RDF resources stored in the object. The following values are valid:

- 0 – URI reference
- 1 – literal
- 2 – blank node
- 3 – number decoded from literal
- 4 – date decoded from literal
- 5 – time decoded from literal

The third field *datatype* is used only for typed literals. In such case this field holds a handle referencing to another *RDFResource* object, which defines the type of the typed literal. If the object is not a typed literal, then this field is initialized to the value *nil*. Untyped literals are regarded as strings.

The fourth field *decoded* holds the decoded string value in the external format. The external format is string representation of a data object. For example all integer, float, and decimal numbers will be decoded to a string in a way that en encoder can reconstruct the internal object. More sophisticated objects, such as those storing dates or times will be decoded to special patterns in the N-Triples format.

The fifth field *encoded* holds data encoded into the internal format. If the encoder function is not present for the required type, then this field will contain the same object as the *decoded* field.

The last field *lang* holds language information. This field is only relevant to literals which have a language defined. In all other cases, this field would hold just an empty string.

4.1.2. Registering custom type to Amos II

After the above structure is defined, the new type needs to be registered to the Amos II system by calling following C function:

```
rdfresource = a_definetype("rdfr", dealloc_rdfr, print_rdfr);
```

The first parameter of the `a_definetype` function is the internal name of the new type. The second and third parameters are deallocation and printing functions. The returned integer is an internal identifier of the new type. It is needed for knowing whether given object is of *RDFResource* type.

After defining the type in C it also needs to be defined in ALisp in order to map an Amos II type named *RDFResource* to the internal name used in C (here *rdfr*). This is done using the ALisp function *createliteraltype*. The first parameter is a string which defines what type name is going to be used in AmosQL. The second one is an optional type being a parent of the new defined type in the type hierarchy of Amos II. The third parameter is a symbol holding the internal type name registered with the C function *a_definetype*. The last parameter is the name of a function which returns the Amos II data type for a given structure.

```
(createliteraltype 'RDFResource '(Object) 'rdfr nil 'rdfr-typefn)
```

4.1.3. Internal functions

There are additional functions that can be added to the global *type table*. Two such functions are *hash* and *compare* functions.

```
typefns[rdfresource].hashfn = rdfr_hash;
```

```
typefns[rdfresource].comparefn = rdfr_compare;
```

The hash function is used to get a unique hash key for every *RDFResource* object. If two objects store the same value, the hash function applied to these objects returns the same values.

The compare function is used to compare two *RDFResource* objects. It takes two objects as parameters and returns *-1*, *0* or *1* if first *RDFResource* object is less, equal, or greater than second.

Amos II also supports reading and creating objects from files. For this purpose a *reader* function is defined. It takes parameters specified in internal system format and creates an *RDFResource* object. The reader function must also be registered to Amos II system using `type_reader_function` function.

```
type_reader_function("RDFR", read_rdfr);
```

This registration function takes two parameters. The first one is a type tag of a linearized object. The second parameter is C reader function for the particular tag.

4.1.4. Creating RDFResource object

After the reader function has been registered, new *RDFResource* objects are created by reading a string such as:

```
#[RDFR 1 "literal string" "en-us" nil]
```

The custom type tag for *rdfr* is followed by a type identifier. Afterwards follows the actual data of the new *RDFResource* object. If the object is a string and has a language defined, then the language string comes after the data. The last parameter is the RDF datatype. It is a URI reference specified in the same notation as the reader function understands. This way of creating new object uses reader function which was registered previously.

There are also AmosQL and ALisp functions defined to create new *RDFResource* objects. These functions take exactly the same parameters as a reader function.

```
AmosQL: rdfr(1, "literal string", "en-us", nil);
ALisp: (rdfr-make 1 "literal string" "en-us" nil)
```

When creating new objects data can be specified either as string representation or by providing object in specific format. For example, these two functions will create identical objects.

```
rdfr(3, "22.2", "", nil); // both encoded to internal real
rdfr(3,   22.2, "", nil);
```

The difference is that when executing first ‘stringified’ function call, the encoder function will be executed in order to encode the string representation to the internal format. By contrast, when executing the second function call, the decoder will be executed to generate the string representation of the passed object. The string representation will be later used when printing the data.

Internal objects can also be used to create literal *RDFResource* objects of kind *1*. In such cases the datatype must be present in order to determine which encoder/decoder should be called.

```
set :datatype = rdfr(0, "http://www.w3.org/2001/XMLSchema#float", "", nil);
rdfr(1, "22.2", "", :datatype); // valid
rdfr(1,   22.2, "", :datatype); // valid
rdfr(1,   22.2, "",       nil); // invalid, generates error
rdfr(1, "22.2", "",       nil); // valid, but will not be encoded to internal object
```

Because the `rdfr()` function is multi-directional, it is very easy to get all the necessary data from the *RDFResource* object using the same function. Also there are additional functions provided to accomplish the same task.

4.1.5. Printing

Whenever an *RDFResource* object is printed, such a formatting is used so that the internal reader function could read it. When printing the data, the *decoded* field in the internal object structure is used.

There may be times when lists of several triples need to be printed out in N-Triples format. In such case ALisp function *rdf-print* should be used.

```
(rdf-print my-triple-list "output.txt")
```

The first argument is a list of triples. A triple is just a simple list containing three *RDFResource* objects. This function uses other ALisp functions to break down the list of triples to *RDFResource* objects and then prints them one by one according to N-triple formats using an external function defined in the wrapper.

The textual representaion of two *RDFResource* objects may differ. For example one can represent the same floating point number in many variants:

```
"25.43E2"^^<http://www.w3.org/2001/XMLSchema#float>
"254.3E1"^^<http://www.w3.org/2001/XMLSchema#float>
```

When an *RDFResource* object is read from the file, it is encoded to a unique internal format. Many different string representations will encode to the same internal object, thus making the numbers above equal.

4.2. Encoders and decoders

In order to integrate the wrapper as seamless as possible, literal types are encoded into internal Amos II objects. This approach allows to use all the built in Amos II functions for the custom types.

To allow easy addition of new types a *type map table* is used. This table contains an RDF datatype URI represented as a string and a kind value together with pointers to encoder and decoder functions. One can easy find required encoder and decoder functions by the kind and the datatype.

The type table is an array of following C struct.

```
struct rdfr_type
{
  int kind;
  oidtype (*encodefn)(oidtype);
  oidtype (*decodefn)(oidtype);
  char *dt_uri;
};
```

The first field in the struct `rdfr_type` holds the identifier for the kind of an object for which the encoder and decoder is associated. The last field `dt_uri`, which is a C string, corresponds to the *datatype*. Either of these fields depending what data is available will be used to match *encoder* and *decoder* to the given object.

The second and third fields are pointers to the encoder and decoder functions. The encoder function has to accept an object containing string representation of data, encode it to an internal Amos II object, and return a handle to that object. The decoder function has to do the opposite, i.e. to accept handle to an object and return a handle to string containing the string representation of a given object.

The index of the *type table map* array is only used internally to distinguish between different combinations of *kind, encoder/decoder* functions and *datatypes.*

Following is the table that provides encoder and decoder functions for *URI*s and *literals* with *float* and *date* datatypes.

```
rdfr_type_t rdfr_type_table[RDFR_TYPES_COUNT] =
{
  {0, e_uri, d_uri, NULL},
  {3, e_fl, d_fl, "http://www.w3.org/2001/XMLSchema#float"},
  {4, e_dt, d_dt, "http://www.w3.org/2001/XMLSchema#date}
};
```

For example, when creating new *RDFResource* object like this:

```
rdfr(3, "22.2", "", nil);
```

the datatype is not needed. The *type map table* will be searched for the *kind* which is 3. Then *e_dt* function will be executed to encode string representation of number to internal object *REAL*.

The introduction of a new type to the type map table basically means writing encoder and decoder functions for the new type and adding them to the type map table. This makes the desing easily extensible.

Encoded object is also used for comparison and hashing. Numbers cannot be hashed or compared using their string representations. The same number can have many different string representations.

Another advantage of the *type map table* is that it allows to dynamically call special routines whenever a new *RDFResource* is created. For example specifing encoder and decoder functions for a *URI* kind means that these functions will be executed whenever new *URI* is created. In such case one can easily implement a hierarchial type structure with the base type of *URI*. Every such custom created element should be appropriately initialized in decoder or encoder functions by setting its *decoded, encoded* and *kind* values.

4.3. External AmosQL functions

There are C functions defined to create and manipulate *RDFResource* objects. These functions are also defined as external Amos II functions [10] implemented in C. Every such external function must be registered to Amos II system using *a_extimpl()* C function, which takes a new function name to be used in AmosQL as a first parameter and a C function definition as a second parameter.

```
a_extimpl("rdfr-data", rdfr_data_fn);
```

Registered external functions also need to be defined in AmosQL before usage. This is achieved using regular AmosQL syntax in a script for defining new functions and adding special implementation specification which says that this new function is external:

```
create function rdfr_data(RDFResource n)->Object as foreign
"rdfr-data";
```

The foreign function `ntriples(Charstring filename)` is the main function used to parse N-Triples streams and emit *RDFResources* to Amis II.

First the C implementation has to know the actual parameters passed to the function. The parameters are retrieved using special functions provided by the foreign function interface. In order to get a string passed as the first parameter the `a_arg` function call should be used:

```
  oidtype a_arg(a_callcontext cxt, int num)
```

Then `ntriples` function calls `NTRparse()` which is a parser generated C function. When this function has done executing three global variables *RDFResource1*, *RDFResource2* and *RDFResource3* will point to the RDF resources extracted from the triple.

Finally the function `ntriples` emits *RDFResource* objects into the Amos II system.

```
a_bind(cxt, 2, RDFResource1);

a_bind(cxt, 3, RDFResource2);

a_bind(cxt, 4, RDFResource3);

a_result(cxt);
```

A result tuple is available in the Amos II system for further processing as soon as it has been emitted, before the foreign function is done executing.

All external functions to be used in AmosQL are defined in an initializing AmosQL script file. Some of the functions are special, multi-directional functions. The defined multi-directional function *rdfr* is used either to create new *RDFResource* objects using the data given as parameters or to extract data from a given *RDFResource* object.

```
create function rdfr(Integer kind, Object data, Charstring
lang, Object datatype)->

  RDFResource n as multidirectional

    ("bbbbf" foreign "rdfrbbbbf")

    ("ffffb" foreign "rdfrffffb");
```

The special signature *bbbbf* means that if the first four parameters are known (bound) and the last parameter (the result) is to be computed (free), then the foreign function in C with the name *rdfrbbbbf* should be called. If four parameters are unknown and the result is known (*ffffb*), then the foreign function in C with the name *rdfrffffb* is called.

4.4. External ALisp functions

ALisp functions can also be defined as external functions implemented in C. Such external ALisp functions are registered using the C function *extfunctionN()*, which takes takes a new function name to be used in ALisp as a first parameter and a C function definition as a second parameter. The *N* specifies how many arguments the registered function is going to take.

```
extfunction1("rdfr-data", rdfr_data_alisp_fn);
```

There are several external ALisp functions defined in the wrapper. Most of them provide the same functionality as the corresponding AmosQL foreign functions. Alisp foreign functions are often used in a testing and developing environments due to the ability to write scripts, which can be very powerful.

The testing script which tests the integrity of the wrapper and the Amos II system is written in ALisp. This script can be easily extended to cover and test more wrapper functionality as the wrapper gets more complicated.

4.5. The lexer

The Flex lexer tokenizes given input by regular expression rules. It looks for patterns which represent data needed for the parser. These patterns are:

- strings representing URI‘s
- strings representing literals
- strings representing names of anonymous nodes

All of the above strings start and end with a special characters mentioned in section *2.1*. If any of these characters are found, a signal is sent to the parser by returning a value representing which character has just been found. If any of mentioned strings are found, they are copied into a global buffer, which is a global array and is shared among the lexer and the parser.

The lexer also utilises states. States are needed because different types of strings can contain different kinds of characters. If a character is found which is a start of a one kind of string, then the lexer enters into the defined state and scans only for a characters from a specified set. When a character which denotes the end of a string is found, the lexer enters into its initial state.

4.6. The parser

The parser calls the lexer function as long as it doesn‘t report end of file. The lexer function returns values that represent the tokens that have been found. Whenever the parser has all the data representing a resource it creates an *RDFResource* object. The newly created object is stored until three of the *RDFResource* objects are created forming a triple. The triple is emitted to the Amos II kernel and the memory is freed afterwards.

5. Evaluation

There N-Triples wrapper should conform to the N-Triples stream format with good performance.

5.1. Conformance

The N-Triples wrapper successfully parsed the test file from official RDF tests page [11]. Therefore, if a given file contains correct N-Triples, the parser will successfully parse it.

The wrapper is also able to output a given triple stream to an output stream, for example a regular file.

The output files were compared with the input files to verify that reading and printing of RDF triples are compatible. All the tests were automatically performed by test scripts. These scripts can be executed whenever something has been changed in Amos II or in the wrapper itself to test if the N-Triples wrapper is still compatible with the Amos II system.

5.2. Performance

The N-Triples wrapper has good performance. It parses 120 000 triples in 11 seconds, i.e. about 11 000 triples per second. Tests indicate that average data read speed is 1.13 megabytes per second. This test was performed on a computer with Intel Pentium(R) processor running at 2.0 GHz.

6. Summary and conclusions

The N-triples wrapper provides a query interface to RDF linked data. It makes possible to query any data source producing data in N-triples format. It is for the user to decide and explore opportunities that the *Semantic Web* provides when using this wrapper.

The N-Triples project has evolved from being just a simple N-Triples parser to a flexible and scalable RDF utility. The wrapper offers seamless integration with Amos II by encoding data into internal system formats. It allows not only to parse and search N-Triple streams, but also to create own RDF resource data types and output them as streams.

The project can be continued by providing compatibility for new literal data types. This can be easily achieved by expanding the *type mapping table*.

The project can also be a reference for upcoming projects as an example of how external AmosQL and ALisp interfaces can be used. It also provides quite a few examples how AStorage system is used with custom objects.

7. Examples

The wrapper can be used in many situations. To select the subject *RDFResource* object from a file of N-Triples the following query could be used:

```
select v[2] from Vector v
  where v in ntriples("nt/w3.nt");
```

The data of any *RDFResource* object can also be accessed in the same query while reading a file. In order to do so, one has to use functions defined in the wrapper which allow to access different kind of properties of *RDFResource* objects, for example:

```
select rdfr_data(o)
  from RDFResource s, RDFResource p, RDFResource o
  where <s, p, o> = ntriples("nt/test.nt");
```

RDFResource objects can be created not only from streams. The following example demosntrates also the multi-directional function capabilities.

```
select data + data
  from Integer kind,
       Real data,
       Charstring lang,
       Object datatype
  where rdfr(kind, data, lang, datatype) =
      rdfr(1, "22.2", "",
  rdfr(0, "http://www.w3.org/2001/XMLSchema#decimal", "", "")
          );
```

Such a query would return *44.4*. Notice how the string value was decoded to a number and the *plus* operation was executed not on the string representation but on the internal object, thus returning correct result.

8. References

1. **Staffan Flodin, Martin Hansson, Vanja Josifovski, Timour Katchaounov, Tore Risch, Martin Sköld, Erik Zeitler.** Amos II Release 12 User's Manual. [Online] November 3, 2009. http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html.
2. **World Wide Web Consortium.** Sematic Web. [Online] http://www.w3.org/standards/semanticweb/.
3. **Tim Berners-Lee**. Notation3 (N3) A readible RDF syntax. [Online] http://www.w3.org/DesignIssues/Notation3.

4. **Dave Beckett, Art Barstow.** N-Triples. [Online] http://www.w3.org/2001/sw/RDFCore/ntriples/
5. **Tore Risch.** ALisp v2 User's Guide. [Online] September 27, 2009. http://user.it.uu.se/~torer/publ/alisp2.pdf.
6. **World Wide Web Consortium.** RDF Test Cases. [Online] February 10, 2004. http://www.w3.org/TR/rdf-testcases/#ntrip_grammar.
7. **Free Software Foundation, Inc.,.** Bison - GNU parser generator. [Online] http://www.gnu.org/software/bison/.
8. **The Flex Project.** flex: The Fast Lexical Analyzer. [Online] http://flex.sourceforge.net/.
9. **Tore Risch.** AStorage a main-memory storage manager. [Online] September 3, 2009. http://user.it.uu.se/~torer/publ/aStorage.pdf.
10. **Tore Risch**. Amos II External Interfaces. [Online] May 31, 2007. http://user.it.uu.se/~torer/publ/external.pdf.
11. **World Wide Web Consortium.** Test file with a variety of legal N-Triples. [Online] October 6, 2003. http://www.w3.org/2000/10/rdf-tests/rdfcore/ntriples/test.nt.
12. **World Wide Web Consortium.** RDF/XML Syntax Specification (Revised). [Online] February 10, 2004. http://www.w3.org/TR/REC-rdf-syntax/

9. Table of figures