3 Access method based on NL-addressing

Abstract

This chapter is aimed to introduce a new access method based on the idea of NL-addressing.

For practical implementation of NLA we need a proper model for database organization and corresponded specialized tools. Hash tables and tries give very good starting point. The main problem is that they are designed as structures in the main memory which has limited size, especially in small desktop and laptop computers. Because of this we need analogous disk oriented database organization.

To achieve such possibilities, we decided to use "Multi-Domain Information Model" (MDIM) and corresponded to it software tools. MDIM and its realizations are not ready to support NL-addressing. We will upgrade them for ensuring the features of NL-addressing via new access method called NL-ArM.

The program realization of NL-ArM, based on specialized hash functions and two main functions for supporting the NL-addressing, access will be outlined. In addition, several operations aimed to serve the work with thesauruses and ontologies as well as work with graphs, will be presented.

3.1 Example of NL-addressing via burst tries

Analyzing Figure 17 and Figure 18, one may see a common structure in both figures. It is *a trie which leafs are containers*. In Figure 17 leafs are Social Security Numbers (SS#) and in Figure 18 leafs are Binary Search Trees (BST). In addition, Figure 14 looks as it is created from many connected Perfect Hash Tables (PHT).

An inference from this is the idea about a multi-way burst trie which:

- Nodes are PHT with entries for all letters of alphabet plus some additional symbols, for instance "0" and "1" ("true" and "false");
- Containers may hold subordinated burst tries.

To illustrate this as a way for possible realization of NL-addressing using burst tries, let see an example (Figure 19).

In this example we use natural numbers instead of letters i.e. their machine codes. This way our alphabet will consists of:

- 256 natural numbers if we will use ASCII encoding;
- 2^{16} natural numbers for UNICODE 16 encoding;

– 2^{32} natural numbers for UNICODE 32 encoding.

We assume that the computer word is 32 bits long and our numbering will permit 2^{32} numbers. What encoding will be used depends of concrete requirements. For easy reading, here we will consider ASCII encoding. This way we will use a small part of all possibilities of such construction.

As we pointed above, we extend the idea of burst tries by creating a hierarchy. In this case every container of a burst trie may hold:

– A string of letters, i.e. a word or phrase of words (such container is colored in magenta on Figure 19 and has number 114);
– Subordinated burst trie.

The path to the container colored in magenta on Figure 19 is A = (66, 101, 101, 114).

It means that:

– Container 66 of root burst trie holds non empty burst trie in which:
  - Container 101 holds non empty burst trie in which:
    - Container 101 holds non empty burst trie in which:
      - Container 114 is holds any information (string).

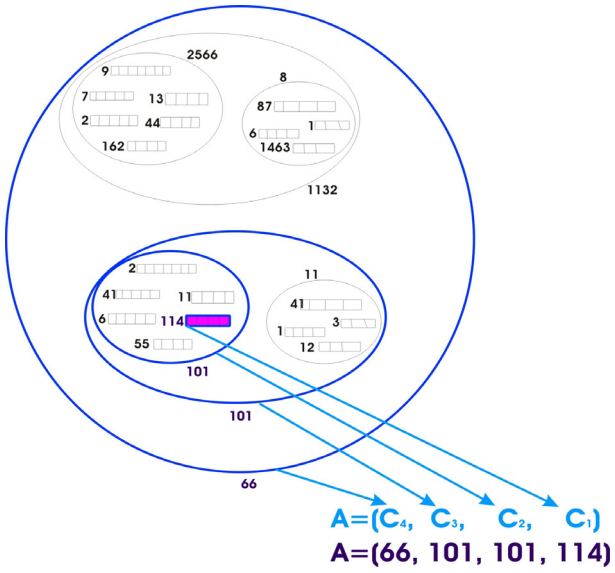

Figure 19. Example of location A=(66,101,101,114)

The numbering is unique for every burst trie. Because of this, there is no problem to have the same numbers of the containers in the subordinated burst tries what is illustrated at the Figure 19 – container 101 holds burst trie with container 101.

Consider the path we just have seen. If we assume these numbers as ASCII codes:

66 = “B”, 101 = “e”, 114 = “r”,

we may “*understand*” the path as the word “***Beer***” (Figure 20).

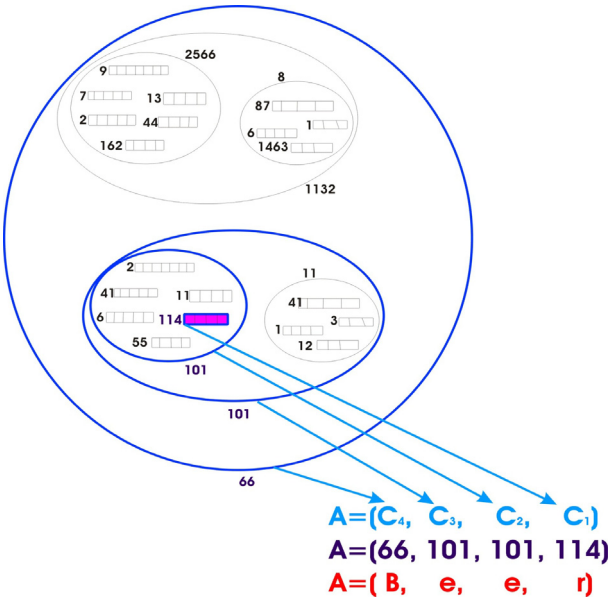

Figure 20. Example of natural language path A=(Beer)

At the end, the container, colored in magenta at Figure 21 and located by this path, may hold arbitrary long string of letters (words, phrases). In our example we choose it to be the remarkable aphorism of Benjamin Franklin:

"Beer is proof that God loves us and wants us to be happy".

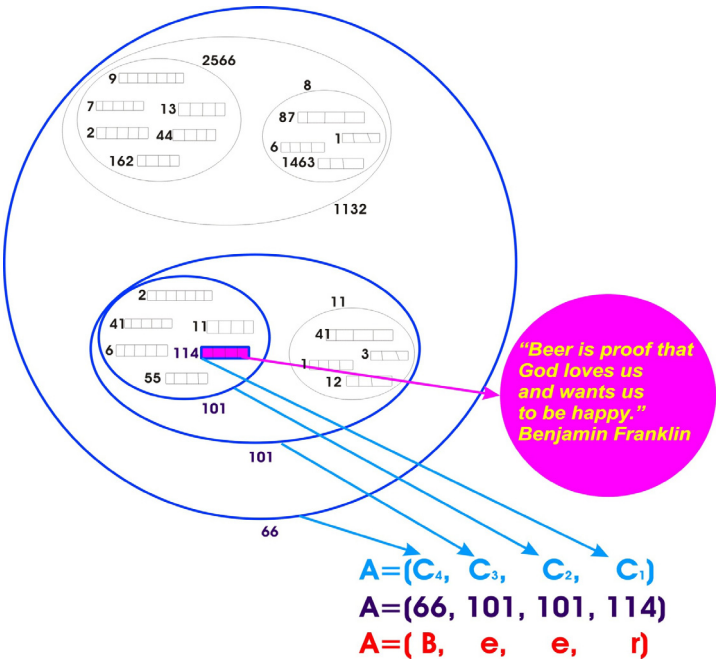

Figure 21. Example of content located by path "Beer"

Perfect hash tables and burst tries give very good starting point. The main problem is that they are designed as *structures in the main memory* which has limited size, especially in small desktop and laptop computers.

3.2 Multi-domain access method "ArM32"

For practical implementation aimed to store very large perfect hash tables and burst tries *in the external memory* (hard disks) we need realization in accordance to the real possibilities. The existing models, analyzed in this research, do not support such structures. Because of this, we decide to make experiments with "Multi-Domain Information Model" (MDIM) [Markov, 1984] and corresponded to it software tools. We will use MDIM as a model for database organization and corresponded specialized tools we will upgrade to our case.

During the last three decades, MDIM has been discussed in many publications. See for instance [Markov et al, 1990; Markov, 2004; Markov et al, 2013].

The program realizations of MDIM are called Multi-Domain Access Method (MDAM) or Archive Manager (ArM) (Table 9).

Table 9. Realizations of MDAM:

no.	name	year	machine	type	language and	operating system
0	**MDAM0**	1975	MINSK 32	37 bit	Assembler	Tape OS
1	**MDAM1**	1981	IBM 360	32 bit	FORTRAN	DOS 360
2	**MDAM2**	1983	PDP 11	16 bit	FORTRAN	DOS 11
3	**MDAM3**	1985	PDP 11	16 bit	Assembler	DOS 11
4	**MDAM4**	1985	Apple II	8 bit	UCSD Pascal	Disquette OS
5	**MDAM5**	1986	IBM PC	16 bit	Assembler, C	MS DOS
6	**MDAM6**	1988	SUN	32 bit	C	SUN UNIX
7	**ArM7**	1993	IBM PC	16 bit	Assembler	MS DOS 3
8	**ArM8**	1998	IBM PC	16 bit	Object Pascal	MS Windows 16 bit
9	**ArM32**	2003	IBM PC	32 bit	Object Pascal	MS Windows 32 bit
10	**NL-ArM**	2012	IBM PC	32 bit	Object Pascal	MS Windows 32 bit
11	**BigArM**	2015 ... under developing		64 bit	Pascal, C, Java	MS Windows, Linux, Cloud

All projects of MDAM and ArM had been done by Krassimir Markov. The program realizations had been done by:

- Krassimir Markov (MDAM0, MDAM1, MDAM2, MDAM3);
- Dimitar Guelev (MDAM4);
- Todor Todorov (MDAM5 written on Assembler with interfaces to PASCAL and C, MDAM5 rewritten on C for IBM PC);
- Vasil Nikolov (MDAM5 interface for LISP, MDAM6);

- Vassil Vassilev (ArM7 and ArM8);
- Ilia Mitov and Krassimira Minkova Ivanova (ArM 32);
- Vitalii Velychko (ArM32 interface to Java);
- Krassimira Borislavova Ivanova (NL-ArM).

For a long period, MDIM has been used as a basis for organization of various information bases.

One of the first goals of the development of MDIM was representing the digitalized military defense situation, which is characterized by a variety of complex objects and events, which occur in the space and time and have a long period of variable existence [Markov, 1984]. The great number of layers, aspects, and interconnections of the real situation may be represented only by information spaces' hierarchy. In addition, the different types of users with individual access rights and needs insist on the realization of a special tool for organizing such information base.

Over the years, the efficiency of MDIM is proved in wide areas of information service of enterprise managements and accounting. For instance, the using MDIM permits omitting the heavy work of creating of OLAP structures [Markov, 2005].

In this research we will use the Archive Manager – "ArM32" developed for MS Windows (32 bit) [Markov, 2004; Markov et al, 2008] and its upgrade to NL-ArM.

The ArM32 elements are organized in numbered information spaces with variable levels. There is no limit for the levels of the spaces. Every element may be accessed by a corresponding multidimensional space address (coordinates) given via coordinate array of type cardinal. At the first place of this array, the space level needs to be given. Therefore, we have two main constructs of the physical organizations of ArM32 information bases – numbered information spaces and elements.

The ArM32 Information space (IS) is realized as a (*perfect*) *hash table stored in the external memory*. Every IS has 2^{32} entries (elements) numbered from 0 up to 2^{32}-1. The number of the entry (element) is called its *co-ordinate*, i.e. the co-ordinate is a 32 bit integer value and it is the number of the entry (element) in the IS.

Every entry is connected to a container with variable length from zero up to 1G bytes. If the container holds zero bytes it is called "empty". In other words, in ArM32, the length of the element (string) in the container may vary from 0 up to 1G bytes. There is no limit for the number of containers in an archive but their total length plus internal indexes could not exceed 2^{32} bytes in a single file.

If all containers of an IS hold other IS, it is called "*IS of corresponded level"* depending of the depth of including subordinated IS. If containers of given IS hold arbitrary information but not other IS, it is called "*Terminal IS*".

To locate a container, one has to define ***the path in hierarchy*** using a ***co-ordinate array*** with all numbers of containers starting from the one of the *root* information space up to the *terminal* information space which is owner of the container.

The hierarchy of information spaces may be not balanced. In other words, it is possible to have branches of the hierarchy which have different depth.

In ArM32, we assume that ***all possible*** *information spaces* ***exist****.*

If all containers of the information space are empty, it is called "***empty****"*.

Usually, most of the ArM32 information spaces and containers are empty. "Empty" means that corresponded structure (space or container) does not occupy disk space. This is very important for practical realizations.

Remembering that **Trie** is a tree for storing strings in which there is one node for every common prefix and the strings are stored in extra leaf nodes, we may say the ArM32 has analogous organization and *can be used to store (burst) tries*.

➢ *Functions of ArM32*

ArM32 is realized as set of functions which may be executed from any user program. Because of the rule that all structures of MDIM exist, we need only two main functions with containers (elements):

- Get the value of a container (as whole or partially);
- Update a container (with several variations).

Because of this, the main ArM32 functions with information elements are:

- *Arm Read* (reading a part or a whole element);
- *Arm Write* (writing a part or a whole element);
- *Arm Append* (appending a string to an element);
- *Arm Insert* (inserting a string into an element);
- *Arm Cut* (removing a part of an element);
- *Arm Replace* (replacing a part of an element);
- *Arm Delete* (deleting an element);
- *Arm Length* (returns the length of the element in bytes).

MDIM operations with information spaces are over:

- **Single space** – *clearing the space*, i.e. updating all its containers to be empty;
- **Two spaces** – there exist several such type of operations. The most used is copying of one space in another, i.e. copying the contents of containers of the first space in the containers of the second. Moving and comparing operations are available, too.

The corresponded ArM32 functions over the spaces are:

- *ArmDelSpace* (deleting the space);
- *ArmCopySpace* and *ArmMoveSpace* (copying/moving the first space in the second in the frame of one file);
- *ArmExportSpace* (copying one space from one file to the other space, which is located in another file).

The ArM32 functions, aimed to serve the navigation in the information spaces return the space address of the **next** or **previous, empty** or **non-empty** elements of the space starting from any given co-ordinates. They are *ArmNextPresent, ArmPrevPresent, ArmNextEmpty*, and *ArmPrevEmpty*.

The ArM32 function, which create indexes, is *ArmSpaceIndex* – returns the space index of the non-empty structures in the given information space.

The service function for counting non-empty ArM32 elements or subspaces is *ArmSpaceCount* – returns the number of the non-empty structures in given information space.

Using ArM32 engine practically we have great limit for the number of dimensions as well as for the number of elements on given dimension. The boundary of this limit in the current realization of ArM32 engine is 2^{32} for every dimension as well as for number of dimensions. Of course, another limitation is the maximum length of the files, which depends on the possibilities of the operating systems and realization of ArM. For instance, in the next version, ArM64 called "BigArM", these limits will be extended to cover the power of 64 bit addressing.

ArM32 engine supports multithreaded concurrent access to the information base in real time. Very important characteristic of ArM32 is possibility not to occupy disk space for empty structures (elements or spaces). Really, only non-empty structures need to be saved on external memory.

Summarizing, the advantages of the ArM32 are:

- Possibility to build growing space hierarchies of information elements;
- Great power for building interconnections between information elements stored in the information base;
- Practically unlimited number of dimensions (this is the main advantage of the numbered information spaces for well-structured tasks, where it is possible "***to address, not to search***").

3.3 NL-ArM access method

MDAM and respectively ArM32 are not ready to support NL-addressing. We have to upgrade them for ensuring the features of NL-addressing. The new access method is called **NL-ArM** (Natural Language Addressing Archive Manager).

The program realization of NL-ArM is based on a specialized hash function and two main functions for supporting the NL-addressing access.

In addition, several operations were realized to serve the work with thesauruses and ontologies as well as work with graphs.

➢ *NL-ArM hash function*

The NL-ArM hash function is called "*NLArmStr2Addr*". It converts a string to space path. Its algorithm is simple: four ASCII symbols or two UNICODE 16 symbols form one 32 bit co-ordinate word. This reduces the space' level four, respectively – two, times. The string is extended with leading zeroes if it is needed. UNICODE 32 does not need converting – one such symbol is one co-ordinate word.

There exists a reverse function, "*NLArmAddr2Str*". It converts space address in ASCII or UNICODE string. The leading zeroes are not included in the string.

The functions for converting are not needed for the end-user because they are used by the NL-ArM upper level operations given below.

All NL-ArM operations access the information by NL-addresses (given by a NL-words or phrases). Because of this we will not point specially this feature.

Let's illustrate the algorithm of NL-ArM hash function.

In the case of Figure 21, the couple

{(name), (content)}

is:

{(B, e, e, r), ("Beer is proof that God loves us and wants us to be happy." Benjamin Franklin)}.

To access the text, we have to convert NL-path (B, e, e, r) to path of numbers (66, 101, 101, 114), i.e. we have the consequence:

Beer => (B, e, e, r) => (66, 101, 101, 114) =>
=> ("Beer is proof that God loves us and wants us to be happy" Benjamin Franklin).

NL-addressing means that human or program will set the correspondence:

(Beer) => ("Beer is proof that God loves us and wants us to be happy" Benjamin Franklin)

and all rest work has be done by the ***NL-ArM hash function*** which has to *convert name in a space path*, i.e.

Beer => (B, e, e, r) => (66, 101, 101, 114).

This hash function is one-one, and because of this, the resulting hash table is a perfect one.

Note that the NL-ArM is an access method and it receives commands only from other program units. Some of them are aimed to serve the human-computer interface and may redirect users' requests directly to NL-ArM (such units are experimental modules presented in this monograph). Other units may concern some information processing like reasoning and may send to NL-ArM requests according theirs need.

In the last case, the programs set the correspondence (name) => (content).

➢ *NL-ArM operations with terminal containers*

Terminal containers are those which belong to terminal information spaces. They hold strings up to 1GB long.

There are two main operations with strings of terminal containers:

- *NLArmRead* – read from a container (all string or substring);
- *NLArmWrite* – update the container (all string or substring).

Additional operations are:

- *NLArmAppend* (appending a substring to string of the container);
- *NLArmInsert* (inserting a substring into string of the container);

- *NLArmCut* (removing a substring from the string of the container);
- *NLArmReplace* (replacing a substring from the string of the container);
- *NLArmDelete* (empting the container);
- *NLArmLength* (returns the length of the string in the container in bytes).

In general, the container may be assumed not only as up to 1GB long string of characters but as some other information again up to 1GB. As a rule, the access methods do not interpret the information which is transferred to and from the main memory. It is important to have possibility to access information in the container as a whole or as set of concatenated parts.

Assuming that all containers exist but some of them are empty, we need only two main operations:

1) To update (write) the string or some of its parts.
2) To receive (read) the string or some of its parts.

The additional operations are modifications of the classical operations with strings applied to this case.

To access information from given container, NL-ArM needs the path to this container and buffer from or to which the whole or a part of its content will be transferred. Additional parameters are length in bytes and possibly - the starting position of substring into the string. When string has to be transferred as a whole, the parameters are the length of the string and zero as number of the starting position.

- ***NL-ArM operations with information spaces (hash tables)***

With information spaces we may provide service operations with hash tables such as counting empty or non-empty containers, copying or moving strings of substrings from containers one to those of another terminal information space. We will not use these operations in the frame of this work.

3.4 Example of NL-storing the Sample graph

As final example of this chapter, let see how the sample graph from previous chapter can be stored using NL-addressing.

To make sample graph more “realistic”, we will put a question about representing the characteristics of the nodes and edges. At the Figure 13 characteristics have been written as comments to nodes and edges.

In the graph, the characteristics of nodes (viz. age, type) may be represented as additional loop edges of type “has_characteristics” and different characteristics may be given by keywords and corresponded values for these edges.

The characteristics of edges (viz. since) may be represented as additional information to the node pointed by the corresponded edge. This information may be given again by corresponded keywords and theirs values.

The final multi-layer representation of our sample graph is given in Table 10 and the final version of the sample graph is shown at Figure 22.

Table 10. Final multi-layer representation of sample graph

	space path		
human location	*Alice*	*Bob*	*Chess*
NL-ArM location	(65, 108, 105, 99, 101)	(66, 111, 98)	(67, 104, 101, 115, 115)
layer (file name)			
has_characteristics	**Alice; Age: 18**	**Bob; Age: 22**	**Chess; Type: Group**
knows	**Bob - since: 2001/10/03**	**Alice - since: 2001/10/04**	
members			**Alice; since: 2005/07/01; Bob; since: 2011/02/14**
is_member	**Chess; since: 2005/07/01**	**Chess; since: 2011/02/14**	

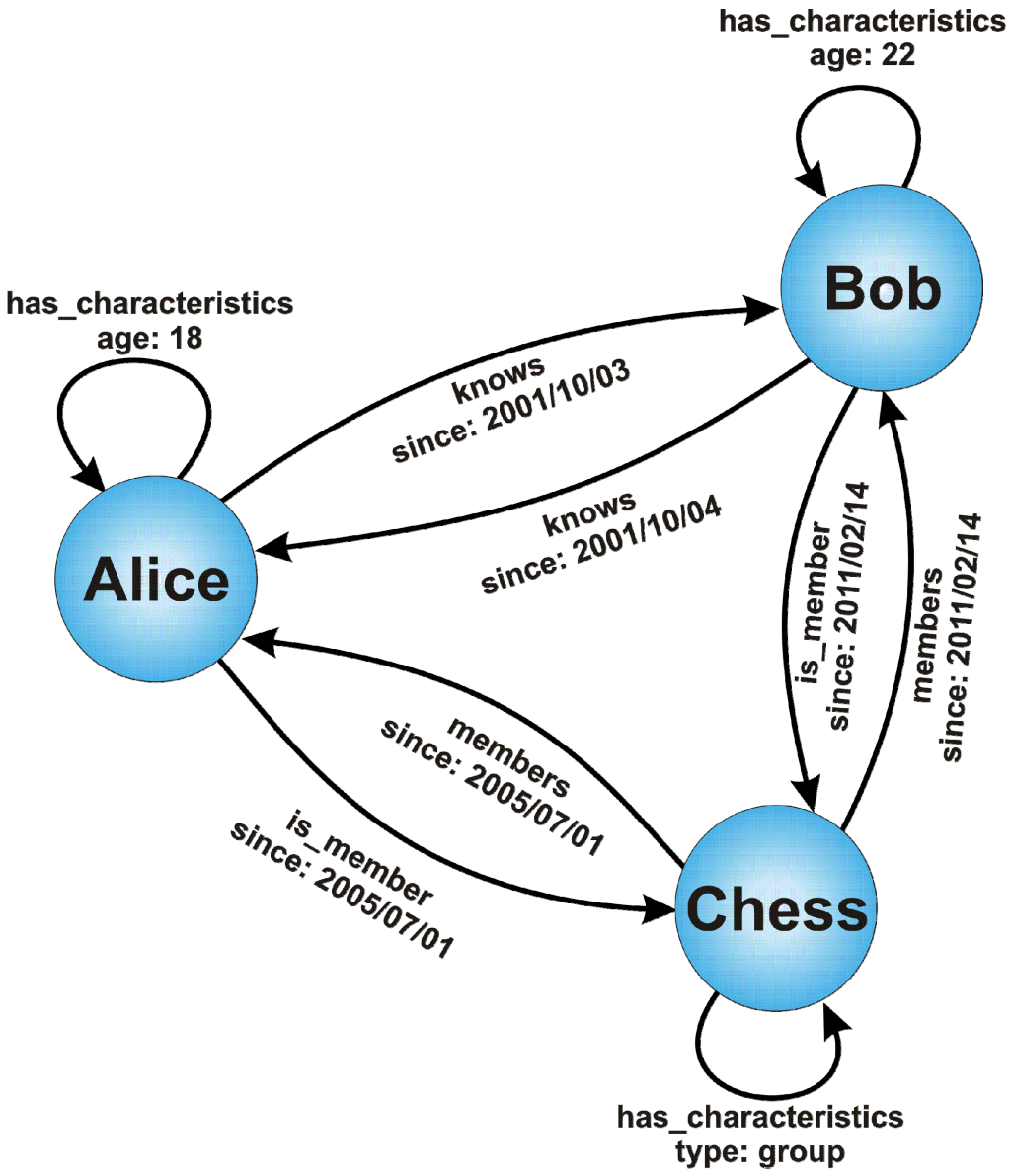

Figure 22. Final variant of the sample graph

Really, the Table 10 show what we will have in our sample database:

- Every layer (row of Table 10) is separate trie and will be stored in a separate file with name of the layer;
- Human locations are given by names: Alice, Bob, and Chess, and NL-ArM (internal computer) locations age given by paths: (65, 108, 105, 99, 101), (66, 111, 98), (67, 104, 101, 115, 115);
- All cells of Table 10 written in bold are containers which hold corresponded information (strings) from the cells;
- The locations (space paths) are common for all layers.

➢ *Storing RDF-graphs by NL-ArM*

We may easy represent the Table 10 by RDF-triples and vice versa (Table 11).

Table 11. Representation of the sample graph by RDF-triples

Subject	*Relation*	*Object*
Alice	*has_characteristics*	**Alice – Age : 18**
Alice	*knows*	**Bob – since : 2001/10/03**
Alice	*is_member*	**Chess – since : 2005/07/01**
Bob	*has_characteristics*	**Bob – Age : 22**
Bob	*knows*	**Alice – since : 2001/10/04**
Bob	*is_member*	**Chess – since : 2011/02/14**
Chess	*has_characteristics*	**Chess –Type : Group**
Chess	*members*	**Alice; Bob**

In other words, NL-ArM is ready for storing RDF information. Mapping of Table 11 in Table 10 is just the algorithm for creating RDF-triple stores based on MDIM and NL-addressing.

From Table 11 it follows that we may define two main information models for storing RDF-graphs using NL-ArM.

The first model we will denote as

RSO model, i.e. Relation-Subject-Object model,

and the second one as

SRO model, i.e. Subject-Relation-Object model.

The first information model for storing RDF-graphs is based on choosing the *relations as separate layers* (file names) and subjects as NL-paths in all layers, i.e.

RSO model: **Relation (Subject) => Object**.

The second is the dual one – the *subjects may be chosen as layers* and the relations as NL-path, i.e.

SRO model: **Subject (Relation) => Object**.

In both cases, the object is the only information to be stored in the archives.

The abstract structure of both models is:

NL-ArM_archive_file_name (NL-address) => Stored_information

What model has to be preferred depends of the sets of relations and subjects, i.e. one that has less size is preferable to be selected as set of layers. If both of models have great size than the next Universal model may be preferred.

In the Universal information model (**UNL model**), both Subject and Relation are equally presented. In this model concatenation of Subject and Relation is assumed as NL-path in a common archive (trie), i.e.

UNL model: **NL-ArM_archive (Subject, Relation) => Object**.

➢ *Conclusion of chapter 3*

This chapter was aimed to introduce a new access method based on the idea of Natural Language Addressing.

MDIM and its realizations are not ready to support NL-addressing. We upgraded them for ensuring the features of NL-addressing via new access method called ***NL-ArM****.*

The program realization of NL-ArM is based on specialized hash functions and two main functions for supporting the NL-addressing access.

In addition, several operations were realized to serve the work with thesauruses and ontologies as well as work with graphs.

NL-ArM is ready for storing RDF information. It is possible to define tree information models for storing RDF-graphs using NL-ArM: (1) ***RSO model*** *(Relation-Subject-Object model), (2)* ***SRO model*** *(Subject-Relation-Object model), and (3)* ***UNL model*** *(****(Subject, Relation) => Object*** *Universal model).*