A SURVEY OF MATHEMATICAL AND INFORMATIONAL FOUNDATIONS OF THE BIGARM ACCESS METHOD

Krassimira Ivanova

Abstract: *The BigArM is an access method for storing and accessing Big Data. It is under development. In this survey we present its mathematical and informational foundations as well as its requirements to realization characteristics. Firstly, we outline the needed basic mathematical concepts, the Names Sets, and hierarchies of named sets aimed to create a specialized model for organization of information bases called "Multi-Domain Information Model" (MDIM). The "Information Spaces" defined in the model are kind of strong hierarchies of enumerations (named sets). Further we remember the main features of hashing and types of hash tables as well as the idea of "Dynamic perfect hashing" and "Trie", especially – the "Burst trie". 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. To solve this problem, dynamic perfect hashing and burst tries will be realized as external memory structures in BigArM.*

Keywords*: BigArM, Big Data, Cloud computing.*

ACM Keywords: *D.4.3 File Systems Management, Access methods.*

Introduction

In this survey we will present the mathematical and informational foundations of a new generation of the access methods based on numbered information spaces [Markov, 1984; Markov, 2004]. Firstly we will outline the needed basic mathematical concepts, the Names Sets, and hierarchies of named sets aimed to create a specialized mathematical model for organization of information bases called "Multi-Domain Information Model" (MDIM). The "Information Spaces" defined in the model are kind of strong hierarchies of enumerations (named sets). Further we will remember the main features of hashing and types of hash tables as well as the idea of "Dynamic perfect hashing" and "Trie", especially – the "Burst trie". 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. To solve this problem, in BigArM dynamic perfect hashing and burst tries will be realized as external memory structures.

Basic mathematical concepts

Let remember the some basic mathematical concepts needed for this research [Bourbaki, 1960; Burgin, 2010].

$\varnothing$ is the *empty set*.

If X is a set, then $r \in X$ means that r belongs to X or r is a member of X. If X and Y are sets, then $Y \subseteq X$ means that Y is a subset of X, i.e., Y is a set such that all elements of Y belong to X.

The *union* $Y \cup X$ of two sets Y and X is the set that consists of all elements from Y and from X.

The *intersection* $Y \cap X$ of two sets Y and X is the set that consists of all elements that belong both to Y and to X.

The *union* $\bigcup_{i \in I} X_i$ of sets X_i is the set that consists of all elements from all sets X_i, $i \in I$.

The *intersection* $\bigcap_{i \in I} X_i$ of sets X_i is the set that consists of all elements that belong to each set X_i, $i \in I$.

The *difference* $Y \setminus X$ of two sets Y and X is the set that consists of all elements that belong to Y but does not belong to X.

If X is a set, then 2^X is the *power set* of X, which consists of all subsets of X. The power set of X is also denoted by $\boldsymbol{P}(X)$.

If X and Y are sets, then $X \times Y = \{(x, y); x \in X, y \in Y\}$ is the direct or Cartesian product of X and Y, in other words, $X \times Y$ is the set of all pairs (x, y), in which x belongs to X and y belongs to Y.

Elements of the set X^n have the form $(x_1, x_2, \ldots, x_n)$ with all $x_i \in X$ and are called n-tuples, or simply, tuples.

A fundamental structure of mathematics is *function*. However, functions are special kinds of binary relations between two sets.

A *binary relation* T between sets X and Y is a subset of the direct product $X \times Y$. The set X is called the *domain* of T ($X = \mathrm{Dom}(T)$) and Y is called the *codomain* of T ($Y = \mathrm{CD}(T)$). The *range* of the relation T is $\mathrm{Rg}(T) = \{y; \exists x \in X ((x, y) \in T)\}$.

The *domain of definition* of the relation T is $\mathrm{DDom}(T) = \{x; \exists y \in Y ((x, y) \in T)\}$. If $(x, y) \in T$, then one says that the elements x and y are in relation T, and one also writes $T(x, y)$.

Binary relations are also called multivalued functions (mappings or maps).

Y^X is the set of all mappings from X into Y.

$X^n = \underbrace{X \times X \times \ldots X \times X}_{n}$.

A *function* (also called a *mapping* or *map* or *total function* or *total mapping*) *f* from *X* to *Y* is a binary relation between sets *X* and *Y* in which:

- There are no elements from X which are corresponded to more than one element from Y;
- To any element from X, some element from Y is corresponded.

Often total functions are also called everywhere defined functions. Traditionally, the element *f*(*a*) is called the image of the element *a* and denotes the value of *f* on the element *a* from *X*. At the same time, the function *f* is also denoted by $f: X \rightarrow Y$ or by *f*(*x*). In the latter formula, *x* is a variable and not a concrete element from *X*.

A *partial function* (or *partial mapping*) *f* from *X* to *Y* is a binary relation between sets *X* and *Y* in which there are no elements from *X* which are corresponded to more than one element from *Y*.

Thus, any function is also a partial function. Sometimes, when the domain of a partial function is not specified, we call it simply a function because any partial function is a total function on its domain.

A *multivalued function* (or *mapping*) *f* from *X* to *Y* is any binary relation between sets *X* and *Y*.

$f(x) \equiv a$ means that the function *f*(*x*) is equal to *a* at all points where *f*(*x*) is defined.

Two important concepts of mathematics are the domain and range of a function. However, there is some ambiguity for the first of them. Namely, there are two distinct meanings in current mathematical usage for this concept. In the majority of mathematical areas, including the calculus and analysis, the term "domain of *f*" is used for the set of all values *x* such that *f*(*x*) is defined. However, some mathematicians (in particular, category theorists), consider the domain of a function $f: X \rightarrow Y$ to be *X*, irrespective of whether *f*(*x*) is defined for all *x* in *X*. To eliminate this ambiguity, we suggest the following terminology consistent with the current practice in mathematics.

If *f* is a function from *X* into *Y*, then the set *X* is called the *domain* of *f* (it is denoted by Dom*f*) and *Y* is called the *codomain* of *T* (it is denoted by Codom*f*). The *range* Rg*f* of the function *f* is the set of all elements from *Y* assigned by *f* to, at least, one element from *X*, or formally, $\mathrm{Rg}f = \{y; \exists\, x \in X\ (f(x) = y)\}$. The *domain of definition* DDom*f* of the function *f* is the set of all elements from *X* that related by *f* to, at least, one element from *Y* is or formally, $\mathrm{DDom}f = \{x; \exists\, y \in Y\ (f(x) = y)\}$. Thus, for a partial function *f*(*x*), its domain of definition DDom*f* is the set of all elements for which *f*(*x*) is defined.

Taking two mappings (functions) f: $X \to Y$ and g: $Y \to Z$, it is possible to build a new mapping (function) gf: $X \to Z$ that is called *composition* or *superposition* of mappings (functions) f and g and defined by the rule $gf(x) = g(f(x))$ for all x from X.

An n-ary relation R in a set X is a subset of the n^{th} power of X, i.e., $R \subseteq X^n$. If $(a_1, a_2, \ldots, a_n) \in R$, then one says that the elements $a_1, a_2, \ldots, a_n$ from X are in relation R.

Named sets

The concept "Named set" was defined by Mark Burgin. Here we will follow [Burgin, 2011].

Named set **X** is a triple **X** = (X, μ, I) where:

- X is the *support* of **X** and is denoted by S(**X**);
- I is the *component of names* (also called *set of names* or *reflector*) of **X** and is denoted by N(**X**);
- μ: $X \to I$ is the *naming map* or *naming correspondence* (also called *reflection*) of the named set **X** and is denoted by n(**X**).

The most popular type of named sets is a named set **X** = (X, μ, I) in which X and I are sets and μ consists of connections between their elements. When these connections are set theoretical, i.e., each connection is represented by a pair (x, a) where x is an element from X and a is its name from I, we have a *set theoretical named set*, which is binary relation.

A name $a \in I$ is called *empty* if $\mu^{-1}(a) = \varnothing$.

A named set **X** is called:

- *Normalized* if in **X** there are no empty names;
- *Conormalized* if in **X** there no elements without names;

Named sets as special cases include:

- Usual sets;
- Fuzzy sets;
- Multisets;
- Enumerations;
- Sequences (countable as well as uncountable)

etc.

A lot of examples of named sets we may find in linguistics studying semantical aspects that are connected with applying different elements of language (words, phrases, texts) to their meaning [Burgin & Gladun, 1989; Burgin, 2011].

A named set **Y** = (Y, η, J) is called *named subset* of named set **X** if $Y \subseteq X$, $J \subseteq I$, and $\eta = \mu\,|_{(Y,J)}$ ($\eta \subseteq \mu \cap (Y \times J)$). In this case **Y** and **X** are connected by the relation of the inclusion.

An ordered tuple of named sets $\Theta = [\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_k]$ where for all i=1, ..., k-1 the condition $N(\mathbf{X}_i) \cap S(\mathbf{X}_{i+1}) \neq \varnothing$ is fulfilled is called *chain of named sets*.

The number k is called a length of the chain Θ.

A tuple of named sets $\Xi_1 = [\mathbf{X}, \mathbf{Y}_1, \mathbf{Y}_2, ..., \mathbf{Y}_n]$ where for all i=1,...,n the condition $N(\mathbf{Y}_i) \cap S(\mathbf{X}) \neq \varnothing$ is fulfilled is called *one level hierarchy of named sets*.

If $N(\mathbf{Y}_i) \cap N(\mathbf{Y}_j) = \varnothing$ and $N(\mathbf{Y}_i) \subseteq S(\mathbf{X})$ for all i=1,...,n, j=1,...,n than Ξ is a *strong one level hierarchy of named sets*.

A tuple of named sets $\Xi_2 = [\mathbf{X}, \Xi_{1,1}, \Xi_{1,2}, ..., \Xi_{1,m}]$ where *sub-hierarhyies* $\Xi_{1j} = [\mathbf{Y}_j, \mathbf{Z}_1, \mathbf{Z}_2, ..., \mathbf{Z}_k]$, j=1,...,m are one level hierarchy of named sets is called *second level hierarchy of named sets*.

If Ξ_{1j}, j=1,...,m, are strong one level hierarchyies of named sets than Ξ_2 is a *strong second level hierarchy of named sets*.

A tuple of named sets $\Xi_n = [\mathbf{X}, \Xi_{n-1,1}, \Xi_{n-1,2}, ..., \Xi_{n-1,l}]$ where $\Xi_{n-1,i}$, i=1,...,l are n-1 level hierarchyies of named sets than Ξ_n is a *n-th level hierarchy of named sets*..

If all sub-hierarhyies of Ξ_n are strong hierarchyies of named sets than Ξ_n is a *strong n-th level hierarchy of named sets*.

Multi-domain information model (MDIM)

We will use strong hierarchies of named sets to create a specialized mathematical model for new kind of organization of information bases. The “Information Spaces” defined in the model are kind of strong hierarchies of enumerations (named sets).

The independence of dimensionality limitations is very important for developing new software systems aimed to process large volumes of high-dimensional data. To achieve this, we need information models and corresponding access methods to cross the boundary of the dimensional limitations and to obtain the possibility to work with large information spaces with variable and practically unlimited number of dimensions. A step in developing such methods is the **Multi-domain Information Model (MDIM)** introduced in [Markov, 1984; Markov, 2004]. Below we remember its main structures and operations.

Basic structures of MDIM

Main structures of MDIM are *basic information elements, information spaces, indexes* and *meta-indexes,* and *aggregates*. The definitions of these structures are given below:

- **Basic information elements**

The basic information element (*BIE*) of MDIM is an arbitrary long string of machine codes (bytes). When it is necessary, the string may be parceled out by lines. The length of the lines may be variable.

- **Information spaces**

Let the universal set ***UBIE*** be the set of all *BIE*.

Let $\boldsymbol{E}_1$ be a set of basic information elements. Let μ_1 be a function, which defines a biunique correspondence between elements of the set $\boldsymbol{E}_1$ and elements of the set $\boldsymbol{C}_1$ of positive integer numbers, i.e.:

$$\boldsymbol{E}_1 = \{e_i \mid e_i \in \boldsymbol{UBIE}\ ,\ i=1,\ldots,\ m_1\}.$$

$$\boldsymbol{C}_1 = \{c_1 \mid c_i \in N,\ i=1,\ldots,m_1\}$$

$$\mu_1\ E_1 \leftrightarrow C_1$$

The elements of $\boldsymbol{C}_1$ are said to be numbers (co-ordinates) of the elements of E_1.

The triple $\boldsymbol{S_1 = (E_1, \mu_1, C_1)}$ is said to be a ***numbered information space of level 1*** (one-dimensional or one-domain information space).

The triple $\boldsymbol{S_2 = (E_2, \mu_2, C_2)}$ is said to be a ***numbered information space of level 2*** (two-dimensional or multi-domain information space of level two) iff the elements of $\boldsymbol{E}_2$ are numbered information spaces of level one (i.e. belong to the set $\boldsymbol{NIS}_1$) and μ_2 is a function which defines a biunique correspondence between elements of $\boldsymbol{E}_2$ and elements of the set C_2 of positive integer numbers, i.e.:

$$\boldsymbol{E}_2 = \{e_i \mid e_i \in \boldsymbol{NIS_1}\ ,\ i=1,\ldots,\ m_2\}.$$

$$C_2 = \{c_i \mid c_i \in N,\ i=1,\ldots,m_2\}$$

$$\mu_2 : E_2 \leftrightarrow C_2$$

The triple $\boldsymbol{S_n = (E_n, \mu_n, C_n)}$ is said to be a ***numbered information space of level n*** (n-dimensional or multi-domain information space) iff the elements of $\boldsymbol{E}_n$ are numbered information spaces of level n-1 (set $\boldsymbol{NIS}_{n-1}$) and μ_n is a function which defines a biunique correspondence between elements of $\boldsymbol{E}_n$ and elements of the set C_n of positive integer numbers, i.e.:

$$\boldsymbol{E}_n = \{e_j \mid e_j \in \boldsymbol{NIS_{n-1}}\ ,\ j=1,\ldots,\ m_n\}.$$

$$C_n = \{c_j \mid c_j \in N,\ j=1,\ldots,m_n\}$$

$$\mu_n : E_n \leftrightarrow C_n$$

Every basic information element "e" is considered as an ***information space S_0*** of level 0. It is clear that the information space $S_0 = (E_0, \mu_0, C_0)$ is constructed in the same manner as all others:

- The machine codes (bytes) b_i, i=1,…,m_0 are considered as elements of E_0;
- The position p_i (natural number) of b_i in the string *e* is considered as co-ordinate of b_i, i.e.

$$C_0 = \{p_k \mid p_k \in N, k=1,\ldots,m_0\},$$

- Function μ_0 is defined by the physical order of b_i in *e* and we have $\mu_0 : E_0 \leftrightarrow C_0$.

This way, the string S_0 may be considered as a set of ***sub-elements (sub-strings).*** The number and length of the sub-elements may be variable. This option is very helpful but it closely depends on the concrete realizations and it is not considered as a standard characteristic of MDIM.

The information space S_n, which contains all information spaces of a given application is called ***information base*** of level **n**. The concept information base without indication of the level is used as generalized concept to denote all available information spaces. For instance every relation data base may be represented as an ***information base of level 3*** which contains set of two dimensional tables.

- **Indexes and meta-indexes**

The sequence $A = (c_n, c_{n-1},\ldots,c_1)$, where $c_i \in C_i$, i=1, …, *n* is called ***multidimensional space address*** of level ***n*** of a basic information element. Every space address of level ***m*, *m* < *n***, may be extended to space address of level *n* by adding leading *n-m* zero codes. Every sequence of space addresses A_1, A_2, …, A_k**,** where ***k*** is arbitrary positive number, is said to be a ***space index***.

Every index may be considered as a basic information element, i.e. as a string, and may be stored in a point of any information space. In such case, it will have a multidimensional space address, which may be pointed in the other indexes, and, this way, we may build a hierarchy of indexes. Therefore, every index, which points only to indexes, is called ***meta-index***.

The approach of representing the interconnections between elements of the information spaces using (hierarchies) of meta-indexes is called ***poly-indexation.***

- **Aggregates**

Let $G = \{S_i \mid i=1,\ldots,n\}$ be a set of numbered information spaces.

Let $\tau = \{v_{ij} : S_i \rightarrow S_j \mid i=const, j=1,\ldots,n\}$ be a set of mappings of one "main" numbered information space $S_i \in G \mid i=const$, into the others $S_J \in G$, j=1, …, n , and, in particular, into itself.

The couple: $D = (G, \tau)$ is said to be an "***aggregate***".

It is clear, we can build **m** aggregates using the set G because every information space $S_J \in G$, j=1, …, n, may be chosen to be a main information space.

Operations in the MDIM

After defining the information structures, we need to present the operations, which are admissible in the model.

In MDIM, we assume that **all** information elements of **all** information spaces **exist**.

If for any S_i : $E_i = Ø \wedge C_i = Ø$, than it is called ***empty***.

Usually, most of the information elements and spaces are empty. This is very important for practical realizations.

- **Operations with basic information elements**

Because of the rule that all structures exist, we need only two operations with a BIE:

- Updating;
- Getting the value.

For both operations, we need two service operations:

- Getting the length of a BIE;
- Positioning in a BIE.

Updating, or simply – ***writing*** the element, has several modifications with obvious meaning:

- Writing as a whole;
- Appending/inserting;
- Cutting/replacing a part;
- Deleting.

There is only one operation for getting the value of a BIE, i.e. ***read*** a portion from a BIE starting from given position. We may receive the whole BIE if the starting position is the beginning of BIE and the length of the portion is equal to the BIE length.

- **Operations with spaces**

We have only one operation with a **single space** – *clearing (deleting) the space*, i.e. replacing all BIE of the space with Ø (empty BIE). After this operation, all BIE of the space will have zero length. Really, the space is cleared via replacing it with empty space.

We may provide two operations with **two spaces**: (1) *copying* and (2) *moving* the first space in the second. The modifications concern how the BIE in the recipient space are processed. We may have:

- Copy/move with clearing the recipient space;
- Copy/move with merging the spaces.

The first modifications first clear the recipient space and after that provide a copy or move operation.

The second modifications may have two types of processing: destructive or constructive. The ***destructive merging*** may be "conservative" or "alternative". In the conservative approach, the BIE of recipient space remains in the result if it is with none zero length. In the other approach – the BIE from donor space remains in the result. In the ***constructive merging*** the result is any composition of the corresponding BIE of the two spaces.

Of course, the move operation deletes the donor space after the operation.

Special kind of operations concerns the navigation in a space. We may receive the space address of the **next** or **previous, empty** or **non-empty** elements of the space starting from any given co-ordinates.

The possibility to count the number of non empty elements of a given space is useful for practical realizations.

- **Operations with indexes, meta-indexes and aggregates**

Operations with indexes, meta-indexes, and aggregates in the MDIM are based on the classical logical operations – intersection, union, and supplement, but these operations are not so trivial. Because of the complexity of the structure of the information spaces, these operations have two different realizations.

Every information space is built by two sets: the set of co-ordinates and the set of information elements. Because of this, the operations with indexes, meta-indexes, and aggregates may be classified in two main types:

- Operations based only on co-ordinates, regardless of the content of the structures;
- Operations, which take in account the content of the structures.

The operations based only on the co-ordinates are aimed to support information processing of analytically given information structures. For instance, such structure is the table, which may be represented by an aggregate. Aggregates may be assumed as an extension of the relations in the sense of the model of Codd [Codd, 1970]. The relation may be represented by an aggregate if the aggregation mapping is one-one mapping. Therefore, the aggregate is a more universal structure than the relation and the operations with aggregates include those of relation theory. What is the new is that the mappings of aggregates may be not one-one mappings.

In the second case, the existence and the content of non empty structures determine the operations, which can be grouped corresponding to the main information structures: elements, spaces, indexes, and meta-indexes. For instance, such operation is the **projection**, which is the analytically given space index of non-empty structures. The projection is given when some coordinates (in arbitrary positions) are fixed and the other coordinates vary for all possible values of coordinates, where non-empty elements exist. Some given values of coordinates may be omitted during processing.

Other operations are transferring from one structure to another, information search, sorting, making reports, generalization, clustering, classification, etc.

Hashing

A *set abstract data type* (set ADT) is an abstract data type that maintains a set ***S*** under the following three operations:

1. *Insert(x)*: Add the key *x* to the set.
2. *Delete(x)*: Remove the key *x* from the set.
3. *Search(x)*: Determine if *x* is contained in the set, and if so, return a pointer to *x*.

One of the most practical and widely used methods of implementing the set ADT is with *hash tables* [Morin, 2005].

The simplest implementation of such data structure is an ordinary array, where *k*-th element corresponds to key *k*. Thus, we can execute all operations in O(1). It is impossible to use this implementation, if the total number of keys is large [Kolosovskiy, 2009].

The main idea behind all hash table implementations is to store a set of $n = |\boldsymbol{S}|$ elements in an array (the hash table) *A* of length m. In doing this, we require a function that maps any element *x* to an array location. This function is called a *hash function h* and the value *h(x)* is called the *hash value of x*. That is, the element *x* gets stored at the array location *A*[*h(x)*].

The occupancy of a hash table is the ratio $\alpha = n/m$ of stored elements to the length of *A* [Morin, 2005].

We have two cases: (1) $m \geq n$ and (2) $m \leq n$:

- In the first case ($m \geq n$) we may expect so called ***perfect hashing*** where every element may be stored in separate cell of the array. In other words, if we have a collection of n elements whose keys are unique integers in (1, m), where $m \geq n$, then we can store the items in a direct address table, T[m], where T_i is either empty or contains one of the elements of our collection.

- In the second case ($m \leq n$) we may expect so called "***collisions***" when two or more elements have to be stored in the same cell f the array.

If we work with two or more keys, which have the *same hash value*, these keys map to the same cell in the array. Such situations are called *collisions*. There are two basic ways to implement hash tables to resolve collisions:

- Chained hash table;
- Open-address hash table.

In ***chained hash table*** each cell of the array contains the linked list of elements, which have corresponding hash value. To add (delete, search) element in the set we add (delete, search) to corresponding linked list. Thus, time of execution depends on length of the linked lists.

In ***open-address hash table*** we store all elements in one array and resolve collisions by using other cells in this array. To perform insertion we examine some slots in the table, until we find an empty slot or understand that the key is contained in the table. To perform search we execute similar routine [Kolosovskiy, 2009].

The study of hash tables follows two very different lines:

1) Integer universe assumption;
2) Random probing assumption.

Integer universe assumption: All elements stored in the hash table come from the universe $U = \{0,...,u-1\}$. In this case, the goal is to design a hash function $h : U \rightarrow \{0, ..., m-1\}$ so that for each $I \in \{0,...,m-1\}$, the number of elements $x \in \boldsymbol{S}$ such that $h(x) = i$ is as small as possible. Ideally, the hash function h would be such that each element of $\boldsymbol{S}$ is mapped to a unique value in $\{0, ..., m-1\}$.

Historically, the **integer universe assumption** seems to have been justified by the fact that any data item in a computer is represented as a *sequence of bits that can be interpreted as a binary number*.

However, many complicated data items require a large (or variable) number of bits to represent and this make the size of the universe very large. In many applications u is much larger than the largest integer that can fit into a single word of computer memory. In this case, *the computations performed in number-theoretic hash functions become inefficient*. This motivates the second major line of research into hash tables, based on *Random probing assumption*.

Random probing assumption: Each element *x* that is inserted into a hash table is a black box that comes with an infinite random probe sequence $x_0, x_1, x_2, ...$ where each of the x_i is independently and uniformly distributed in {*0, ..., m−1*}.

Both the integer universe assumption and the random probing assumption have their place in practice.

When there is an easily computing mapping of data elements onto machine word sized integers then hash tables for integer universes are the method of choice.

When such a mapping is not so easy to compute (variable length strings are an example) it might be better to use the bits of the input items to build a good pseudorandom sequence and use this sequence as the probe sequence for some random probing data structure [Morin, 2005].

Perfect hash function

We consider hash tables under the *integer universe assumption*, in which the key values *x* come from the universe *U* = {0*, ..., u−1*}. A hash function *h* is a function whose domain is *U* and whose level is the set {*0, ..., m−1*}, $m \leq u$.

A hash function h is said to be a ***perfect hash function*** for a set S ⊆ U if, ***for every x ∈ S, h(x) is unique.***

A *perfect hash function h* for *S* is ***minimal*** if *m* = |*S*|, i.e., *h* is a bisection between *S* and {0, ..., *m* − 1}. Obviously a minimal perfect hash function for *S* is desirable since it allows us to store all the elements of *S* in a single array of length *n*. Unfortunately, perfect hash functions are rare, even for *m* much larger than *n* [Morin, 2005].

The set of elements, *S*, may be:

- *Static* (no updates);
- *Dynamic* where fast queries, insertions, and deletions must be made on a large set.

*"**Dynamic perfect hashing**"* is useful for the second type of situations. In this method, the entries that hash to the same slot of the table are organized as separate *second-level hash table*. If there are *k* entries in this set *S*, the second-level table is allocated with k^2 slots, and its hash function is selected at random from a universal hash function set so that it is *collision-free* (i.e. a perfect hash function). Therefore, the look-up cost is guaranteed to be O(1) in the worst-case [Dietzfelbinger et al, 1994].

Perfect hashing can be used in many applications in which we want to assign a unique identifier to each key without storing any information on the key. One of the most obvious applications of perfect hashing (or k-perfect hashing) is when we have a small fast memory in which we can store the perfect hash function while the keys and associated satellite data are stored in slower but larger memory. The size of

a block or a transfer unit may be chosen so that *k* data items can be retrieved in one read access. In this case we can ensure that data associated with a key can be retrieved in a single probe to slower memory. This has been used for example in hardware routers.

Perfect hashing has also been found to be competitive with traditional hashing in internal memory on standard computers. *Recently perfect hashing has been used* ***to accelerate algorithms on graphs*** *when the graph representation does not fit in main memory* [Belazzougui et al, 2009].

For the purposes of CRP we need possibility to use *perfect hashing with dynamic and very large (practically – unlimited) set, S, of elements with variable length of strings*. In this case, the computing mapping of data elements onto machine word sized integers is not so easy to compute (we have long strings with variable length). In the same time, we could not use the bits of the input items to build a good pseudorandom sequence and use this sequence as the probe sequence for some random probing data structure, because of very large, unlimited, set, *S*, of elements.

Tries

> *"As defined by me, nearly 50 years ago, it is properly pronounced "tree" as in the word "retrieval". At least that was my intent when I gave it the name "Trie". The idea behind the name was to combine reference to both the structure (a tree structure) and a major purpose (data storage and retrieval)".*
>
> *Edward Fredkin, July 31, 2008*

Trie is a tree for storing strings in which there is one node for every common prefix. The strings are stored in extra leaf nodes.

A *trie* can be thought of as an *m*-ary tree, where *m* is the number of characters in the alphabet. A search is performed by examining the key one character at a time and using an *m*-way branch to follow the appropriate path in the trie, starting at the root. In other words, in the *multi-way trie* (Figure 1), each node has a potential child for each letter in the alphabet. Below is an example of a multi-way trie indexing the three words BE, BED, and BACCALAUREATE [Pfenning, 2012].

Tries are distinct from the other data structures because they explicitly assume that the keys are a sequence of values over some (finite) alphabet, rather than a single indivisible entity. Thus tries are particularly well-suited for handling variable-length keys. Also, when appropriately implemented, tries can provide compression of the set represented, because common prefixes of words are combined together; words with the same prefix follow the same search path in the trie [Sahni, 2005].

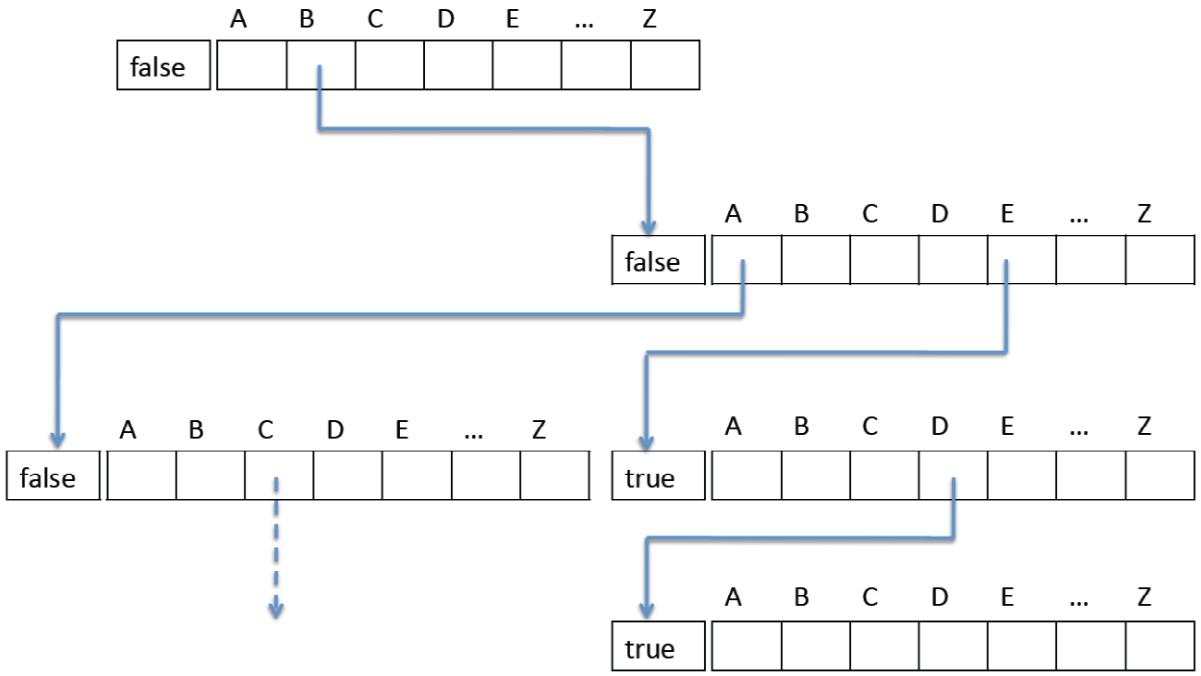

Figure 1. Example of multi-way trie [Pfenning, 2012]

Burst Tries

The tree data structures compared to hashing have three sources of inefficiency [Heinz et al, 2002]:

- First, the average search lengths is surprisingly high, typically exceeding ten pointer traversals and string comparisons even on moderate-sized data sets with highly skew distributions. In contrast, a search under hashing rarely requires more than a string traversal to compute a hash value and a single successful comparison;
- Second, for structures based on Binary Search Trees (BSTs), the string comparisons involved redundant character inspections, and were thus unnecessarily expensive. For example, given the query string "middle" and given that, during search, "Michael" and "midfield" have been encountered, it is clear that all subsequent strings inspected must begin with the prefix "mi";
- Third, in tries the set of strings in a subtrie tends to have a highly skew distribution: typically the vast majority of accesses to a subtrie are to find one particular string. Thus use of a highly time-efficient, space-intensive structure for the remaining strings is not a good use of resources [Heinz et al, 2002].

These considerations led to the burst trie. A ***burst trie*** is an *in-memory* data structure, designed for sets of records that each has a unique string that identifies the record and acts as a key. Formally, a string ***s***

with length *n* consists of a series of symbols or characters c_i for *i*=0;...;*n*, chosen from an alphabet A of size |A|. It is assumed that |A| is small, typically no greater than 256 [Heinz et al, 2002].

A ***burst trie*** consists of three distinct components (Figure 2): a set of records, a set of containers, and an access trie.

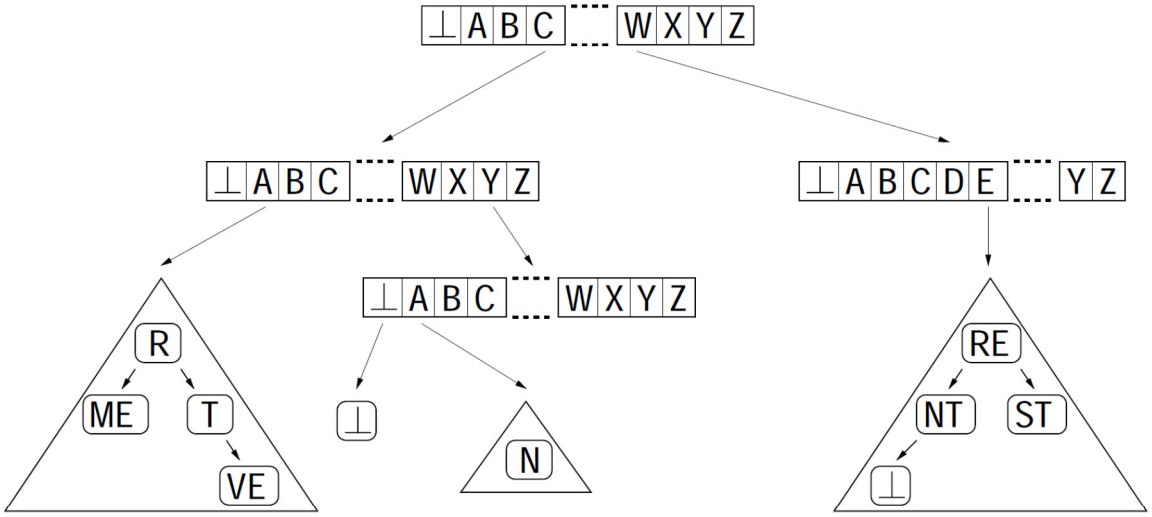

Figure 2. Burst trie with BSTs used in containers [Heinz et al, 2002]

Records. A record contains a string; information as required by the application using the burst trie (that is, for information such as statistics or word locations); and pointers as required to maintain the container holding the record. *Each string is unique*;

Containers. A container is a small set of records, maintained as a simple data structure such as a list or a *binary search tree* (BST). For a container at depth *k* in a burst trie, all strings have length at least k and the first k characters of all strings are identical. It is not necessary to store these first k characters. Each container also has a header, for storing the statistics used by heuristics for bursting. Thus a particular container at depth 3 containing “author” and “automated” could also contain “autopsy” but not “auger”;

Access trie. An access trie is a trie whose leaves are containers. Each node consists of an array *p*, of length |A|, of pointers, each of which may point to either a trie node or a container, and a single empty-string pointer to a record. The |A| array locations are indexed by the characters $c \in A$. The remaining pointer is indexed by the empty string.

The depth of the root is defined to be 1. Leaves are at varying depths.

A burst trie can be viewed as a generalization of other proposed variants of trie.

Figure 2 shows an example of a burst trie storing ten records whose keys are “came”, “car”, “cat”, “cave”, “cy”, “cyan”, “we”, “went”, “were”, and “west” respectively. In this example, the alphabet A is the set of letters from A to Z, and in addition an empty string symbol ⊥ is shown; the container structure used is a BST. In this figure, the access trie has four nodes, the deepest at depth 3. The leftmost container has four records, corresponding to the strings “came”, “car”, “cat”, and “cave”. One of the strings in the rightmost container is “⊥”, corresponding to the string “we”. The string “cy” is stored wholly within the access trie, as shown by the empty-string pointer to a record, indexed by the empty string [Heinz et al, 2002].

Natural Language Addressing

Analyzing Figure 1 and Figure 2, one may see a common structure in both figures. It is *a trie which leafs are containers*. In Figure 1 leafs are Social Security Numbers (SS#) and in Figure 2 leafs are Binary Search Trees (BST). In addition, Figure 1 looks as it is created from many connected Perfect Hash Tables (PHT).

In addition, if we take in account the possibilities of MDIM, we may use for realization a multi-way burst trie which:

- Nodes are PHT with entries for all numbers of given interval, for instance $(0, 2^{32}-1)$;
- Containers may hold subordinated burst tries.

One very important consequence is to use as interval only the numbers which are codes of letters in any encoding system: ASCII, UNICODE16, or UNICODE32. This case is called “Natural Language Addressing” (NL-addressing) [Ivanova et al, 2013; Ivanova, 2014a; Ivanova, 2014b].

The idea of NL-addressing is to use encoding of the name both as relative address and as route in a Multi-dimensional information space and this way to speed the access to stored information. For instance, let have the next definition: “Pirrin: A mountain with co-ordinates (x, y)”. In the computer memory, it may be stored in a file at relative address “50067328” and the corresponded index couple may be: (“Pirrin”, “50067328”). At the memory address “50067328” the main text, “A mountain ... (x,y)” will be stored. To read/write the main text, firstly we need to find name “Pirrin” in the index and after that to access memory address “50067328” to read/write the definition. If we assume that name “Pirrin” in the computer memory is encoded by six numbers (letter codes), for instance by using ASCII encoding system Pirrin is encoded as (80, 105, 114, 114, 105, 110), than we may use these codes for direct address to memory, i.e. (“Pirrin”, “80, 105, 114, 114, 105, 110”).

Above we have written two times the same name as letters and codes. Because of this we may omit this couple and index, and read/write directly to the address “80, 105, 114, 114, 105, 110”. For human this address will be shown as “Pirrin”, but for the computer it will be “80, 105, 114, 114, 105, 110”.

Multi-domain access method “ArM32”

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.

For practical implementation aimed to store very large perfect hash tables and burst tries *in the external memory* (hard disks) we need relization in accordance to the real possibilities. One possible solution is to use “Multi-Domain Information Model” (MDIM) [Markov, 1984] and corresponded to it software tools.

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 the corresponded to MDIM access method and its different program realizations during the years have different names: **M**ulti-**D**omain **A**ccess **M**ethod (MDAM), **Ar**chive **M**anager (ArM), and **N**atural **L**anguage Addressing **Ar**chive **M**anager (NL-ArM), **Big** Data **Ar**chive **M**anager **(BigArM)** (Table 1).

Developing the method and all projects of its realizations 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).

Table 1. 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	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

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].

ArM32

Crrent realization of MDIM, respectively – MDAM, is the Archive Manager – "ArM32" developed for MS Windows (32 bit) [Markov, 2004; Markov et al, 2008] and its upgrate 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 wich 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:

- *ArmRead* (reading a part or a whole element);
- *ArmWrite* (writing a part or a whole element);
- *ArmAppend* (appending a string to an element);
- *ArmInsert* (inserting a string into an element);
- *ArmCut* (removing a part of an element);
- *ArmReplace* (replacing a part of an element);
- *ArmDelete* (deleting an element);
- *ArmLength* (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 firstspace 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.

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***").

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.

- **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.

Requirements to BigArM realization characteristics

Main characteristics of program realizations of MDAM are shown in Table 2.

Using ArM32 engine 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. Main limitation of ArM32 is that the length of archive files may be 4GB long. This cause that in practical implementations we have not so big number of dimensions (usually it is about 200).

What is needed is to extend possibilities of ArM32 from 32 bit up to 64 bit addressing capabilities and to rationalize the internal hash structures to speed access from milliseconds down to microseconds per one access operation. This will be done in ongoing developing of its new version called “BigArM” for 64 bit machines and operating systems like MS Windows and Linux. In addition, BigArM will permit new kind of Cloud processing of Big Data, called “Collect/Report Paradigm” (CRP) [Markov et al, 2014; Markov & Ivanova, 2015].

Table 2. Main characteristics of program realizations of MDAM

№	name	max dimensions	max size of element	max number of elements	max size of archive	max size of information base	access time
0	MDAM0	1	128 bytes	128	512 words	16K words	minutes
1	MDAM1	1	256 bytes	256	1 KB	10 MB	seconds
2	MDAM2	2	256 bytes	2^{31} (10 000)	32 KB	4 MB	seconds
3	MDAM3	2	256 bytes	2^{31} (10 000)	32 KB	4 MB	seconds
4	MDAM4	1	80 bytes	25	30 elements	4KB	deciseconds
5	MDAM5	2	64 KB	2^{31} (1 000 000)	32KB	80 MB	centiseconds
6	MDAM6	2	64 KB	2^{31} (1 000 000)	32KB	90 MB	centiseconds
7	ArM7	2+2	1 GB	2^{60}	4 GB	10 GB	milliseconds
8	ArM8	2+2	1 GB	2^{60}	4 GB	10 GB	milliseconds
9	ArM32	200	1 GB	2^{64} (max 4G)	4 GB	1TB	milliseconds
10	NL-ArM	200	1 GB	2^{64} (max 4G)	4 GB	1TB	milliseconds
11	BigArM	2^{32}	4 GB	2^{64}	1 PB	1 YB	microseconds

Conclusion

In this survey we presented mathematical and informational foundations as well as requirements to realization characteristics BigArM - an access method for storing and accessing Big Data. It is under development. Firstly, we outlined the needed basic mathematical concepts, the Names Sets, and hierarchies of named sets aimed to create a specialized model for organization of information bases

called “Multi-Domain Information Model” (MDIM). The “Information Spaces” defined in the model are kind of strong hierarchies of enumerations (named sets). Further we remembered the main features of hashing and types of hash tables as well as the idea of “Dynamic perfect hashing” and “Trie”, especially – the “Burst trie”. 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. To solve this problem, dynamic perfect hashing and burst tries will be realized as external memory structures in BigArM.

Special attention we have paid to MDIM and its realizations ArM2 and 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.

Finaly, we have presented the main requirements to BigArM realization characteristics. The expected project characteristics of BigArM are sumarized in Table 3.

Table 3. Project characteristics of BigArM

Programming language	Object Pascal, C, Java
Operational environment	Windows, Linux, Cloud
Maximal size of the elements in the archive	**4 GB**
Maximal size of the archive	**2^{64} (>1 PB = 2^{50})**
Maximal sizeof the information base	**no limit (>1 YB = 2^{80})**
Access time	**microseconds**
Dimensions of the information spaces	**variable up to max 2^{32}**
Number of elements in an archive	**2^{64}**
Main technologies for accessing data	– **Direct R/W addressing** – **NL R/W addressing** – **Collect/Report paradigm**

Bibliography

[Belazzougui et al, 2009] Djamal Belazzougui, Fabiano C. Botelho, Martin Dietzfelbinger, "Hash, Displace, and Compress", In: Algorithms - ESA 2009 - 17th Annual European Symposium, Copenhagen, Denmark, September 7-9, 2009, Proceedings. Lecture Notes in Computer Science Volume 5757, Springer, 2009, pp 682-693. DOI 10.1007/978-3-642-04128-0_61 Print ISBN: 978-3-642-04127-3 Online ISBN: 978-3-642-04128-0. http://link.springer.com/chapter/10.1007%2F978-3-642-04128-0_61 (accessed: 20.07.2013).

[Bourbaki, 1960] Bourbaki, N., "Theorie des Ensembles", Hermann, Paris, 1960, English version: Bourbaki, N. Theory of Sets, Volume package: Elements of Mathematics. Springer, 1st ed. 1968, 2nd printing 2004, ISBN 978-3-540-22525-6, 414 p.

[Burgin & Gladun, 1989] Mark Burgin, Victor Gladun, "Mathematical Foundations of Semantic Networks Theory", In: LNCS No.: 364, Springer, 1989. pp. 117-135.

[Burgin, 2010] Mark Burgin, "Theory of Information - Fundamentality, Diversity and Unification", World Scientific Publishing Co. Pte. Ltd. Singapore, 2010, ISBN-13 978-981-283-548-2, 672 p.

[Burgin, 2011] Mark Burgin, "Theory of Named Sets", Nova Science Publishers Inc (United States), 2011, ISBN-13: 9781611227888, 681 p.

[Codd, 1970] Codd, E., "A relation model of data for large shared data banks", Magazine Communications of the ACM, 13/6, 1970, pp. 377-387

[Dietzfelbinger et al, 1994] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan, "Dynamic Perfect Hashing: Upper and Lower Bounds", SIAM J. Comput, 23, 4, 1994, ISSN: 0097-5397, pp. 738-761, http://portal.acm.org/citation.cfm?id=182370# (accessed: 20.07.2013).

[Heinz et al, 2002] Steffen Heinz, Justin Zobel, Hugh E. Williams, "Burst Tries: A Fast, Efficient Data Structure for String Keys", ACM Transactions on Information Systems (TOIS), Volume 20, Issue 2, April 2002, pp. 192 – 223, ACM New York, NY, USA, doi>10.1145/506309.506312, http://dl.acm.org/citation.cfm?id=506312 (accessed: 20.07.2013)

[Ivanova et al, 2013] Krassimira B. Ivanova, Koen Vanhoof, Krassimir Markov, Vitalii Velychko, "Introduction to the Natural Language Addressing", International Journal "Information Technologies & Knowledge" Vol.7, Number 2, 2013, ISSN 1313-0455 (printed), 1313-048X (online), pp. 139–146.

[Ivanova, 2014a] Krassimira Ivanova, “Multi-Layer Knowledge Representation”, International Journal “Information Content and Processing”, Vol. 1, Number 4, 2014, ISSN 2367-5128 (printed), 2367-5152 (online), pp. 303 - 310.

[Ivanova, 2014b] Krassimira Ivanova, “Practical Aspects of Natural Language Addressing”, In: G. Setlak, K. Markov (ed.), Computational Models for Business and Engineering Domains, ITHEA®, 2014, Rzeszow, Poland, Sofia, Bulgaria, ISBN: 978-954-16-0066-5 (printed), ISBN: 978-954-16-0067-2 (online), pp. 172 – 186.

[Kolosovskiy, 2009] Kolosovskiy M., “Simple implementation of deletion from open-address hash table”, Cornell University Library, ArXiv e-prints, 2009, http://adsabs.harvard.edu/abs/2009arXiv0909.2547K (accessed: 20.07.2013)

[Markov & Ivanova, 2015] Markov Kr., Kr. Ivanova, “General Structure of Collect/Report Paradigm for Storing and Accessing Big Data”, Int. J. Information Theories and Applications, 22/3, 2015, pp. 266-290

[Markov et al, 1990] K. Markov, T. Todorov, V. Nikolov, “Multidomain Access Method for the IBM PC”, Research in Informatics, Vol. 3, Academie-Verlag Berlin, 1990, pp. 218-230.

[Markov et al, 2008] Markov, K., Ivanova, K., Mitov, I., & Karastanev, S., “Advance of the access methods”, International Journal of Information Technologies and Knowledge, 2(2), 2008, pp. 123–135.

[Markov et al, 2013] Markov, Krassimir, Koen Vanhoof, Iliya Mitov, Benoit Depaire, Krassimira Ivanova, Vitalii Velychko and Victor Gladun, "Intelligent Data Processing Based on Multi- Dimensional Numbered Memory Structures", Diagnostic Test Approaches to Machine Learning and Commonsense Reasoning Systems, IGI Global, 2013, pp. 156-184, doi:10.4018/978-1-4666-1900-5.ch007, ISBN: 978 1-4666-1900-5, EISBN: 978-1-4666- 1901-2
Reprinted in: Markov, Krassimir, Koen Vanhoof, Iliya Mitov, Benoit Depaire, Krassimira Ivanova, Vitalii Velychko and Victor Gladun, "Intelligent Data Processing Based on Multi-Dimensional Numbered Memory Structures", Data Mining: Concepts, Methodologies, Tools, and Applications, IGI Global, 2013, pp. 445-473, doi:10.4018/978-1-4666-2455-9.ch022, ISBN13: 978-1-4666-2455-9, EISBN13: 978-1-4666-2456-6

[Markov et al, 2014] Kr. Markov, Kr. Ivanova, K. Vanhoof, B. Depaire, V. Velychko, J. Castellanos, L. Aslanyan, St. Karastanev, “Storing Big Data Using Natural Language Addressing”, In: N. Lyutov (ed.), Int. Sc. Conference “Informatics in the Scientific Knowledge”, VFU, Varna, Bulgaria, 2014, ISSN: 1313-4345, pp. 147-164.

[Markov, 1984] Markov Kr., "A Multi-domain Access Method", Proceedings of the International Conference on Computer Based Scientific Research, Plovdiv, 1984, pp. 558-563.

[Markov, 2004] Markov, K., "Multi-domain information model", Int. J. Information Theories and Applications, 11/4, 2004, pp. 303-308

[Markov, 2005] Markov, K., "Building data warehouses using numbered multidimensional information spaces", International Journal of Information Theories and Applications, 12(2), 2005, pp. 193–199

[Morin, 2005] Pat Morin, "Hash tables", Chapter 9, of "Handbook of data structures and applications" /edited by Dinesh P. Mehta and Sartaj Sahni, Chapman & Hall/CRC computer & information science, ISBN 1-58488-435-5, 2005, 1321 p.

[Pfenning, 2012] Frank Pfenning, "Lecture Notes on Tries", Lecture 21, In 15-122: Principles of Imperative Computation November 8, 2012. http://www.cs.cmu.edu/~fp/courses/15122-f12/lectures/21-tries.pdf (accessed: 20.07.2013).

[Sahni, 2005] Sartaj Sahni, "Tries", Chapter 28, of "Handbook of data structures and applications" /edited by Dinesh P. Mehta and Sartaj Sahni, Chapman & Hall/CRC computer & information science, 2005, 1321 pages, ISBN 1-58488-435-5.

Authors' Information

Krassimira Ivanova– *University of National and World Economy, Sofia, Bulgaria; Institute of Mathematics and Informatics, BAS, Bulgaria; e-mail: krasy78@mail.bg*

Major Fields of Scientific Research: Software Engineering, Business Informatics, Data Mining, Multidimensional multi-layer data structures in self-structured systems