# 2 Storing models

***Abstract***

*This chapter is aimed to introduce the main data structures and storing technologies which we will use to compare our results. Mainly they are graph data models as well as Resource Description Framework (RDF) storage and retrieval technologies.*

*Firstly we shortly define concepts of storage model and data model.*

*Mapping of the data models to storage models is based on program tools called "access methods". Their main characteristics will be outlined.*

*During the eighties of the last century, the total growing of the research and developments in the computers' field, especially in image processing, data mining and mobile support cause impetuous progress of establishing convenient "spatial information structures" and "spatial-temporal information structures" and corresponding access methods. Important cases of spatial representation of information are Graph models. Because of this, Graph models and databases will be discussed more deeply and examples of different graph database models will be presented. The need to manage information with graph-like nature especially in RDF-databases has reestablished the relevance of this area.*

*In accordance with this, the analyses of RDF databases as well as of the storage and retrieval technologies for RDF structures will be in the center of our attention. Storing models for several popular ontologies and summary of main types of storing models for ontologies and, in particular, RDF data, will be discussed.*

*At the end of this chapter, our attention will be paid to addressing and naming (labeling) in graphs with regards to introducing the Natural Language Addressing (NL-addressing) in graphs. A sample graph will be analyzed to find its proper representation.*

*Taking in account the interrelations between nodes and edges, we will see that a "multi-layer" representation is possible and the identifiers of nodes and edges can be avoided. As result of the analysis of the example, the advantages and disadvantages of the multi-layer representation of graphs will be outlined.*

*The specialized mathematical model for new kind of organization of information bases called "Multi-Dmain Information Model" (MDIM) and its realizations will be presented.*

## 2.1 Storage model and Data model

Let remember that the "***data storage***" is a part of a computer that stores information for subsequent use or retrieval [AHD, 2009]. It is a device consisting of electronic, electrostatic, electrical, hardware, or other elements into which data may be entered, and from which data may be obtained as desired. For instance it may be magnetic tapes, hard drive storage, network storage, removable media (USB devices, flash drives, SD cards, DVDs), and online storage (Cloud storage [Mell & Grance, 2011]) [Greenwood, 2012].

The "***storage model***" is a model that captures key *physical* aspects of data structure in a data store. The ***storage schema*** (internal schema) is a specification of how the data relationships and rules specified in the logical schema of a database will be mapped to the physical storage level in terms of the available constructs, such as aggregation into records, clustering on pages, indexing, and page sizing and caching for transfer between secondary and primary storage. Storage schema facilities vary widely between different **D**ata**B**ase **M**anagement **S**ystems (DBMS) [Daintith, 2004].

On the other hand, a **data model** is a model that captures key *logical* aspects of data structure in a database, i.e. underlying the structure of a database is a *data model*. A data model is a collection of conceptual tools for describing the real-world entities to be modeled in the database and the relationships among these entities. Data models differ in the primitives available for describing data and in the amount of semantic detail that can be expressed. The various data models that have been proposed fall into three different groups: *object-based* logical models, *record-based* logical models, and *physical* data models. Physical data models are used to describe data at the lowest level. [Silberschatz et al, 1996].

There is multitude of reviews and taxonomies of data models [Silberschatz et al, 1996; Navathe, 1992; Beeri, 1988; Kerschberg et al, 1976]. An evolutionary scheme of the most important and widely accepted DataBase (DB) models is outlined in Figure 10. Rectangles denote database models (db-models), arrows indicate influences, and circles denote theoretical developments. A time-line in years is shown on the left [Angles & Gutierrez, 2008].

From a database point of view, the conceptual tools that make up a db-model should at least address data structuring, description, maintenance, and a way to retrieve or query the data. According to these criteria, a db-model consists of three components [Codd, 1980]:

- A set of data structure types;
- A set of operators or inference rules;
- A set of integrity rules.

Several proposals for db-models only define the data structures, sometimes omitting operators and/or integrity rules [Angles & Gutierrez, 2008].

In addition, each db-model proposal is based on certain theoretical principles, and serves as base for the development of related models.

The short overview of Database Models (db-models) below follows one given in [Angles & Gutierrez, 2008].

Before the advent of the relational model, most db-models focused essentially on the specification of data structures on actual file systems (Figure 10).

At this time the main information structure is the "record". Let remember that the "record" is a logical sequence of fields which contain data eventually connected to unique identifier (a "key"). The identifier (key) is aimed to distinguish one sequence from another [Stably, 1970]. The records are united in the sets, called "files". There exist three basic formats of the records – with *fixed*, *variable* and *undefined* length.

In the *context-free file models*, storing of the records is not connected to their content and depends only on external factors – the sequence, disk address or position in the file. The main idea of *the context-depended file models* is that the part of the record is selected as a key which is used for making decision where to store the record and how to search it. This way the content of the record influences on the access to the record [Markov et al, 2008].

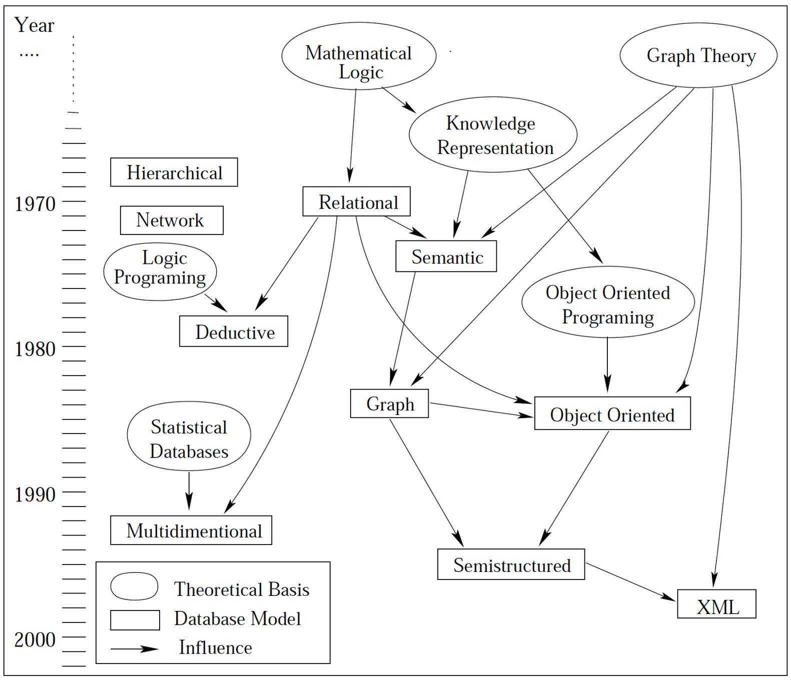


***Figure 10. Evolutionary scheme of DB-models [Angles & Gutierrez, 2008]***

Modern DataBase Management Systems (DBMS) are built using context-depended file models such as: unsorted sequential files with records with keys; sorted files with fixed record length; static or dynamic hash files; index file and files with data; clustered indexed tables [Connolly & Begg, 2002].

Two representative database models are the *hierarchical* [Tsichritzis & Lochovsky, 1976] and the *network* [Taylor & Frank, 1976] models, both of which place emphasis on the physical level.

The *relational db-model* was introduced by Codd [Codd, 1980] and highlights the concept of abstraction levels by introducing the idea of separation between physical and logical levels. It is based on the notions of sets and relations.

As opposed to previous models, *semantic db-models* [Peckham & Maryanski, 1988] allow database designers to represent objects and their relations in a natural and clear manner, providing users with tools to faithfully capture the desired domain semantics. A well-known example is the entity-relationship model [Chen, 1976].

*Object-oriented db-models* [Kim, 1990] appeared in the eighties, when most of the research was concerned with so-called "advanced systems for new types of applications [Beeri, 1988]". These db-models are based on the object-oriented paradigm and their goal is to represent data as a collection of objects, which are organized into classes, and are assigned complex values.

*Graph db-models* made their appearance alongside object-oriented db-models. These models attempt to overcome the limitations imposed by traditional db-models with respect to capturing the inherent graph structure of data appearing in applications such as hypertext or geographic information systems, where the interconnectivity of data is an important aspect. This type of models is outlined further in this text.

*Semi-structured db-models* [Buneman, 1997] are designed to model data with a flexible structure, for example, documents and Web pages. Semi-structured data is neither raw nor strictly typed, as in conventional database systems. These db-models appeared in the nineties. Further in this chapter we will outline such type model called Resource Description Framework (RDF).

Closely related to them is the *XML (eXtensible Markup Language)* [Bray et al, 1998] model, which did not originate in the database community. Although originally introduced as a document exchange standard, it soon became a general purpose model, focusing on information with tree-like structure [Angles & Gutierrez, 2008].

Mapping of the data models to storage models is based on program tools called "*access methods"*.

## 2.2 Memory management and access methods

Memory management is a complex field of computer science. Over the years, many techniques have been developed to make it more efficient [Ravenbrook, 2010]. Memory management is usually divided into three areas: *hardware*, *operating system*, and *applications*, although the distinctions are a little fuzzy. In most computer systems, all three are present to some extent, forming layers between the user's program and the actual memory hardware:

- **Memory management at the hardware level** is concerned with the electronic devices that actually store data. This includes things like RAM, Associative memory, and memory caches [Mano, 1993];
- **Memory in the operating system** must be allocated to user programs, and reused by other programs when it is no longer required. The operating system can pretend that the computer has more memory than it actually does, and that each program has the machine's memory to itself. Both of these are features of *virtual memory* systems;
- **Application memory management** involves supplying the memory needed for a program's objects and data structures from the limited resources available, and recycling that memory for reuse when it is no longer required. Because in general, application programs cannot predict in advance how much memory they are going to require, they need additional code to handle their changing memory requirements.

Application memory management combines two related tasks:

- **Allocation**: when the program requests a block of memory, the memory manager must allocate that block out of the larger blocks it has received from the operating system. The part of the memory manager that does this is known as the *allocator*;
- **Recycling**: when memory blocks have been allocated, but the data they contain is no longer required by the program, the blocks can be recycled for reuse. There are two approaches to recycling memory: either the programmer must decide when memory can be reused (known as *manual memory management*); or the memory manager must be able to work it out (known as *automatic memory management*).

The progress in memory management gives the possibility to allocate and recycle not directly blocks of the memory but structured regions or fields corresponding to some types of data. In such case, we talk about corresponded "*access methods*".

The **Access Methods (AM)** had been available from the beginning of the development of computer peripheral devices. As many devices so many possibilities for developing different AM there exist. Our attention is focused only to the access methods for devices for permanently storing the information with direct access such as magnetic discs, flash memories, etc. [Markov et al, 2008].

In the beginning, the AM were functions of the Operational Systems' Core or so called Supervisor, and were executed via corresponding macro-commands in the assembler languages [Stably, 1970] or via corresponding input/output operators in the high level programming languages like FORTRAN, COBOL, PL/I, etc.

The establishment of the first databases in the sixties of the previous century caused gradually accepting the concepts "physical" as well as "logical" organization of the data [CODASYL, 1971; Martin, 1975]. In 1975, the concepts "access method", "physical organization" and "logical organization" became clearly separated. In the same time Christopher Date [Date, 1977] wrote:

"The DataBase Management System (DBMS) does not know anything about:

a) Physical records (blocks);
b) How the stored fields are integrated in the records (nevertheless that in many cases it is obviously because of their physical disposition);
c) How the sorting is realized (for instance it may be realized on the base of physical sequence, using an index or by a chain of pointers);
d) How is realized the direct access (i.e. by index, sequential scanning or hash addressing).

This information is a part of the structures for data storing but it is used by the access method but not by the DBMS".

Every access method presumes an exact organization of the file, which it is operating with and is not related to the interconnections between the files, respectively, – between the records of one file and that in the others files. These interconnections are controlled by the physical organization of the DBMS [Date, 2004].

Therefore, in the DBMS we may distinguish four levels:

- Basic access methods of the core (supervisor) of the operation system;
- Specialized access methods realized using basic access methods;
- Physical organization of the DBMS;
- Logical organization of the DBMS.

During the eighties of the last century, the total growing of the research and developments in the computers' field, especially in image processing, data mining and mobile support cause impetuous progress of establishing convenient "spatial information structures" and "spatial-temporal information structures" and corresponding access methods. From different points of view, this period has been presented in [Ooi et al, 1993; Gaede & Günther, 1998; Arge, 2002; Mokbel et al, 2003; Moënne-Loccoz, 2005]. Usually, the "one-dimensional" (linear) AM are used in the classical applications, based on the alphanumerical information, whereas the "multi-dimensional" (spatial) methods are aimed to serve the work with graphical, visual, multimedia information [Markov et al, 2013].

## 2.3 Interconnections between raised access methods

Maybe one of the most popular analyses of the genesis of the access methods is given in [Gaede & Günther, 1998]. The authors presented a scheme of the genesis of the basic multi-dimensional AM and theirs modifications. This scheme firstly was proposed in [Ooi et al, 1993] and it was expanded in [Gaede & Günther, 1998]. An extension in direction to the multi-dimensional spatio-temporal access methods was given in [Mokbel et al, 2003].

The survey [Markov et al, 2008] presents a new variant of this scheme, where the new access methods, created after 1998, are added. A comprehensive bibliography of corresponded articles, where the methods are firstly presented, is given.

The access methods, presented on Figure 11 [Markov et al, 2008] may be classified as follow:

- One-dimensional AM:
  - Context free;
  - Context depended;
- Multidimensional Spatial AM:
  - Point AM:
    - Multidimensional Hashing;
    - Hierarchical Access Methods;
    - Space Filling Curves for Point Data;
  - Spatial AM:
    - Transformation;
    - Overlapping Regions;
    - Clipping;
    - Multiple Layers;
- Metric Access Methods;
- High Dimensional Access Methods:
  - Data Approximation;
  - Query Approximation:
    - Clustering of the database;
    - Splitting the database;
- Spatio-Temporal Access Methods:
  - Indexing the past;
  - Indexing the present;
  - Indexing the future.

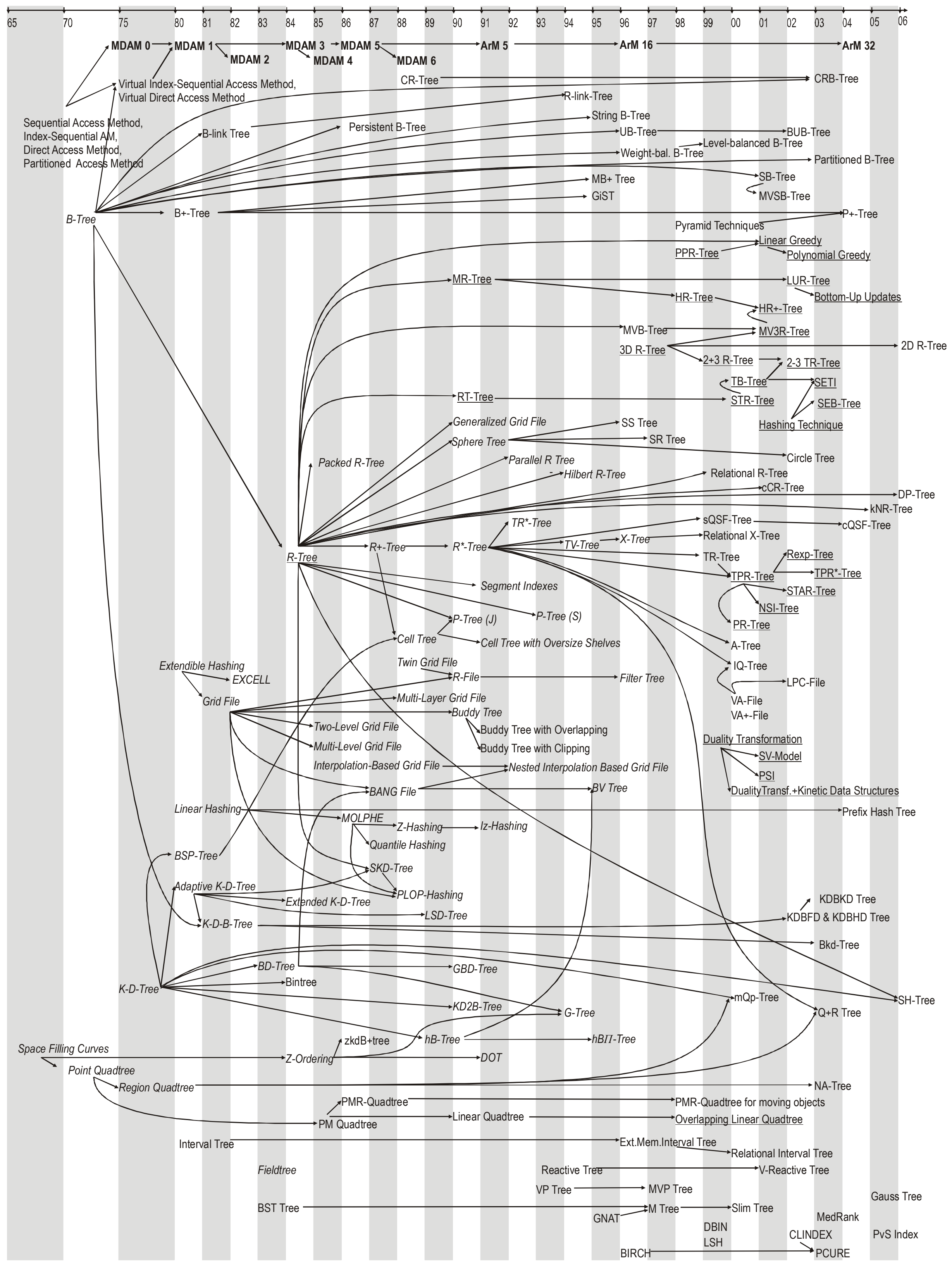


***Figure 11. Genesis of the Access Methods and their modifications extended variant of [Gaede & Günther, 1998; Mokbel et al, 2003] presented in [Markov et al, 2008]***

### ➢ *One-dimensional access methods*

One-dimensional AM are based on the concept "*record*". The "record" is a logical sequence of fields, which contain data eventually connected to unique identifier (a "key"). The identifier (key) is aimed to distinguish one sequence from another [Stably, 1970]. The records are united in the sets, called "*files*". There exist three basic formats of the records – with fixed, variable, and undefined length.

In the ***context-free methods***, the storing of the records is not connected to their content and depends only on external factors – the sequence, disk address, or position in the file. The necessity of stable file systems in the operating systems does not allow a great variety of the context-free AM. There are three main types well known from sixties and seventies: *Sequential Access Method* (SAM); *Direct Access Method* (DAM) and *Partitioned Access Method* (PAM) [IBM, 1965-68].

The main idea of the ***context-depended AM*** is that a part of the record is selected as a key, which is used for making decision where to store the record and how to search it. This way, the content of the record influences the access to the record.

Historically, from the sixties of the previous century on, the attention is directed mainly to the second type of AM. Modern DBMS are built using context-depended AM such as: unsorted sequential files with records with keys; sorted files with fixed record length; static or dynamic hash files; index files and files with data; clustered indexed tables [Connolly & Begg, 2002].

### ➢ *Multidimensional spatial access methods*

Multidimensional Spatial Access Methods are developed to serve information about spatial objects, approximated with points, segments, polygons, polyhedrons, etc. The implementations are numerous and include traditional multi-attributive indexing, geographical and/or information systems for global monitoring for environment and security, spatial databases, content indexing in multimedia databases, etc.

From the point of view of the spatial databases, the access methods can be split into two main classes of access methods – *Point Access Methods* and *Spatial Access Methods* [Gaede & Günther, 1998].

**Point Access Methods** are used for organizing multidimensional point objects. Typical instances are traditional records, where every attribute of the relation corresponds to one dimension. These methods can be separated in three basic groups:

- Multidimensional Hashing (for instance Grid File and its varieties, EXCELL, Twin Grid File, MOLPHE, Quantile Hashing, PLOP-Hashing, Z-Hashing, etc);
- Hierarchical Access Methods (includes such methods as KDB-Tree, LSD-Tree, Buddy Tree, BANG File, G-Tree, hB-Tree, BV-Tree, etc.);
- Space Filling Curves for Point Data (like Peano curve, N-trees, Z-Ordering, etc).

**Spatial Access Methods** are used for working with objects, which have an arbitrary form. The main idea of the spatial indexing of non-point objects is to use an approximation of the geometry of the examined objects as more simple forms. The most used approximation is Minimum Bounding Rectangle (MBR), i.e. minimal rectangle, which sides are parallel of the coordinate axes and completely include the object. There exist approaches for approximation with Minimum Bounding Spheres (SS Tree) or other polytopes (Cell Tree), as well as their combinations (SR-Tree) [Gaede & Günther, 1998].

The usual problem when one operates with spatial objects is their overlapping. There are different techniques to avoid this problem. From the point of view of the techniques for the organization of the spatial objects, Spatial Access Methods can be split in four main groups:

- **Transformation** – this technique uses transformation of spatial objects to points in the space with more or less dimensions. Most of them spread out the space using space filling curves (Peano Curves, z-ordering, Hibert curves, Gray ordering, etc.) and then use some point access method upon the transformed data set;
- **Overlapping Regions** – here the data sets are separated in groups; different groups can occupy the same part of the space, but every space object is associated with only one of the groups. The access methods of this category operate with data in their primary space (without any transformations) eventually in overlapping segments. Methods which use this technique includes R-Tree, R-link-Tree, Hilbert R-Tree, R*-Tree, Sphere Tree, SS-Tree, SR-Tree, TV-Tree, X-Tree, P-Tree of Schiwietz, SKD-Tree, GBD-Tree, Buddy Tree with overlapping, PLOP-Hashing, etc.;
- **Clipping** – this technique uses the clipping of one object to several sub-objects, which will be stored. The main goal is to escape overlapping regions. However this advantage can lead to the tearing of the objects, extending the resource expenses, and decreasing the productivity of the method. Representatives of this technique are R+-Tree, Cell-Tree, Extended KD-Tree, Quad-Tree, etc.;
- **Multiple Layers** – this technique can be considered as a variant of the techniques of Overlapping Regions, because the regions from different layers can overlap. Nevertheless there exist some important differences: first – the layers are organized hierarchically; second – every layer splits the primary space in a different way; third – the regions of one layer never overlaps; fourth – the data regions are separated from the space extensions of the objects. Instances for these methods are Multi-Layer Grid File, R-File, etc.

### ➢ *Metric access methods*

Metric Access Methods deal with relative distances of data points to chosen points, named anchor points, vantage points or pivots [Moënne-Loccoz, 2005]. These methods are designed to limit the number of distance computation, calculating first distances to anchors, and then finding the searched point in a narrowed region. These methods are preferred when the distance is highly computational, as e.g. for the dynamic time warping distance between time series. Representatives of these methods are: Vantage Point Tree (VP Tree), Bisector Tree (BST-Tree), Geometric Near-Neighbor Access Tree (GNNAT), as well as the most effective from this group – Metric Tree (M Tree) [Chavez et al, 2001].

## ➢ *High dimensional access methods*

Increasing the dimensionality strongly aggravates the qualities of the multidimensional access methods. Usually, these methods exhaust their possibilities at dimensions around **15**. Only X-Tree reaches the boundary of **25** dimensions, after which this method gives worse results then sequential scanning [Chakrabarti, 2001].

The exit of this situation is based on the data approximation and query approximation in sequential scan. These methods form a new group of access methods – High Dimensional Access Methods.

**Data approximation** is used in VA-File, VA+-File, LPC-File, IQ-Tree, A-Tree, P+-Tree, etc.

For **query approximation**, two strategies can be used:

- Examine only a part of the database, which is more probably to contain the resulting set – as a rule these methods are based on the clustering of the database. Some of these methods are: DBIN, CLINDEX, PCURE;
- Splitting the database to several spaces with fewer dimensions and searching in each of them. Here two main methods are used:
  - Random Lines Projection. Representatives of this approach are MedRank, which uses B+-Tree for indexing every arbitrary projection of the database, and PvS Index, which consist of combination of iterative projections and clustering.
  - Locality Sensitive Hashing, which is based on the set of local-sensitive hashing functions [Moënne-Loccoz, 2005].

## ➢ *Spatio-temporal access methods*

The Spatio-Temporal Access Methods have additional defined time dimensioning [Mokbel et al, 2003]. They operate with objects, which change their form and/or position during the time. According to position of time interval in relation to present moment, the Spatio-Temporal Access Methods are divided to:

- ***Indexing the past***, i.e. these methods operate with historical spatio-temporal data. The problem here is the continuous increase of the information over time. To overcome the overflow of the data space two approaches are used – sampling the stream data at certain time position or updating the information only when data is changed. Spatio-temporal indexing schemes for historical data can be split in three categories:
  - The first category includes methods that manage spatial and temporal aspects into already existing spatial data;
  - The second category can be explained as snapshots of the spatial information in each time instance;
  - The third category focuses on trajectory-oriented queries, while spatial dimension lag on second priority.

  Representatives of this group are: RT-Tree, 3DR-Tree, STR-Tree, MR-Tree, HR-Tree, HR+-Tree, MV3R-Tree, PPR-Tree, TB-Tree, SETI, SEB-Tree;

- **Indexing the present**. In contrast to previous methods, where all movements are known, here the current positions are neither stored nor queried. Some of the methods, which answer the questions of the current position of the objects are 2+3R-Tree, 2-3TR-Tree, LUR-Tree, Bottom-Up Updates, etc.;
- **Indexing the future**. These methods have to answer the questions about the current and future position of a moving object – here are embraced the methods like PMR-Quadtree for moving objects, Duality Transformation, SV-Model, PSI, PR-Tree, TPR-Tree, TPR*-tree, NSI, VCIR-Tree, STAR-Tree, REXP-Tree.

## 2.4 Structured data models

Traditional database systems rely on the relational data model.

When it was proposed in the early 1970's by Codd, a logician, the relational model generated a true revolution in data management. In this simple model data is represented as relations in first order structures and queries as first order logic formulas. It enabled researchers and implementers to separate the logical aspect of the data from its physical implementation. Thirty years of research and development followed, and they led to today's mature and highly performance relational database systems [Mendelzon et al, 2001].

## 2.5 Semi-structured data models

The age of the Internet brought new data management applications and challenges. Data is now accessed over the Web, and is available in a variety of formats, including HTML, XML, as well as several applications specific data formats. Often data is mixed with free text, and the boundary between data and text is sometimes blurred. The way the data can be retrieved also varies considerably: some instances can be downloaded entirely; others can only be accessed through limited capabilities. To accommodate all forms and kinds of data, the database research community has introduced the ***"semi-structured data model"***, ***where data is self-describing, irregular***, ***and graph-like***. The new model captures naturally Web data, such as HTML, XML, or other application specific formats [Mendelzon et al, 2001].

The topic of semi-structured data is relatively recent [Buneman, 2001]. Applications that manage semi-structured data are becoming increasingly commonplace. Current approaches for storing semi-structured data use existing storage machinery - they either map the data to ***relational databases, or use a combination of flat files and indexes*** [Bhadkamkar et al, 2009].

In semi-structured data, the information that is normally associated with a schema contained within the data, which is sometimes called "self-describing". In some forms of semi-structured data, there is no separate schema, in others it exists but only places loose constraints on the data, Semi-structured data has recently emerged as an important topic of study for a variety of reasons. First, there are data sources such as the Web, which we would like to treat as databases but which cannot be constrained by a schema. Second, it may be desirable to have an extremely flexible format for data exchange between disparate databases. Third, even when dealing with structured data, it may be helpful to view it as semi-structured for the purposes of browsing [Buneman, 2001].

The importance of semi-structured models which are "graph-like" revived the interest to *Graph models*.

### 2.6 Graph models and databases

***Graph database model*** is a model in which the data structures for the schema and/or instances are modeled as a directed, possibly labeled, graph, or generalizations of the graph data structure, where data manipulation is expressed by graph-oriented operations and type constructors, and appropriate integrity constraints can be defined over the graph structure [Angles & Gutierrez, 2008].

*Graph database model* can be defined as those in which data structures for the schema and instances are modeled as graphs or generalizations of them, and data manipulation is expressed by graph-oriented operations and type constructors.

The notion of graph database model can be conceptualized with respect to three basic components, namely:

- Data structures;
- Transformation language;
- Integrity constraints.

Hence, a graph database model is characterized as follows:

- Data and/or the schema are represented by graphs, or by data structures generalizing the notion of graph (hypergraphs or hypernodes) [Guting, 1994; Levene & Loizou, 1995; Kuper & Vardi, 1984; Paredaens et al, 1995; Kunii, 1987; Graves et al, 1995a; Gyssens et al, 1990];
- Data manipulation is expressed by graph transformations, or by operations whose main primitives are on graph features like paths, neighborhoods, subgraphs, graph patterns, connectivity, and graph statistics (diameter, centrality, etc.) [Gyssens et al, 1990; Graves et al, 1995a; Guting, 1994];
- Integrity constraints enforce data consistency. These constraints can be grouped in schema-instance consistency, identity and referential integrity, and functional and inclusion dependencies. Examples of these are: labels with unique names, typing constraints on nodes, functional dependencies, domain and range of properties [Graves et al, 1995b; Kuper & Vardi, 1993; Klyne & Carroll, 2004; Levene & Poulovassilis, 1991].

#### ➢ *Examples of graph database models*

In these examples we illustrate the most representative graph database models and other related models that do not fit properly as graph database models, but use graphs, for example, for navigation, for defining views, or as language representation.

To give a flavor of the modeling in each proposal, we will use as a running example the toy genealogy from [Angles & Gutierrez, 2008] (Figure 12). The genealogy diagram (right-hand side) is represented as two tables (left-hand side) NAME-LASTNAME and PERSON-PARENT (Children inherit the last name of the father just for modeling purposes).

The examples below are divided into two categories:

- Graph models with explicit schema (Table 2);
- Graph models with implicit schema (Table 3).

In the both tables, on the left side of the rectangle the corresponded schema is presented and on the right side of the rectangle the examples of instances are given. For all examples in both tables a brief explanation is added.

| NAME | LASTNAME |
|---|---|
| George | Jones |
| Ana | Stone |
| Julia | Jones |
| James | Deville |
| David | Deville |
| Mary | Deville |

| PERSON | PARENT |
|---|---|
| Julia | George |
| Julia | Ana |
| David | James |
| David | Julia |
| Mary | James |
| Mary | Julia |

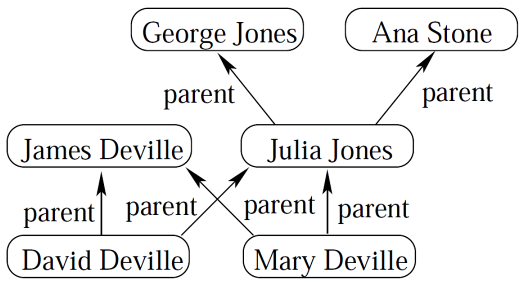
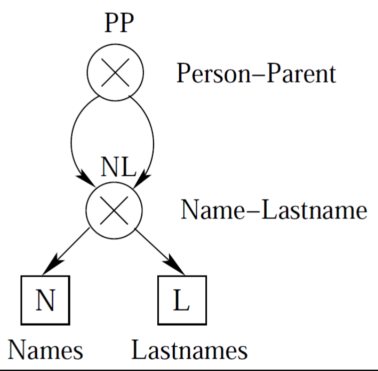


***Figure 12. Running example: the toy genealogy***

***Table 2.*** ***Examples of the graph models with explicit schema***

**Schema**



**Instance**

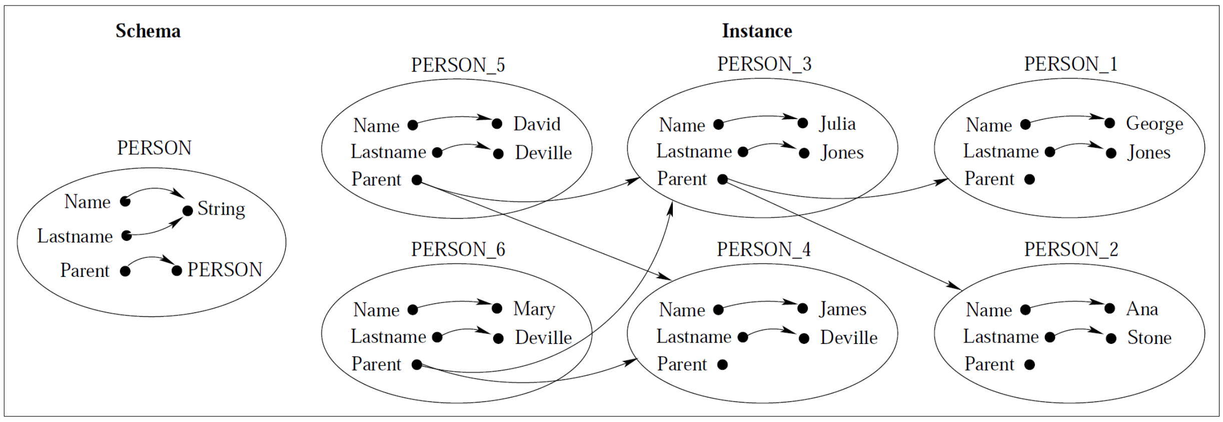
| *I (N)* | |
|---|---|
| *1* | *val (1)* |
| 1 | George |
| 2 | Ana |
| 3 | Julia |
| 4 | James |
| 5 | David |
| 6 | Mary |

| *I (L)* | |
|---|---|
| *1* | *val (1)* |
| 7 | Jones |
| 8 | Stone |
| 9 | Deville |

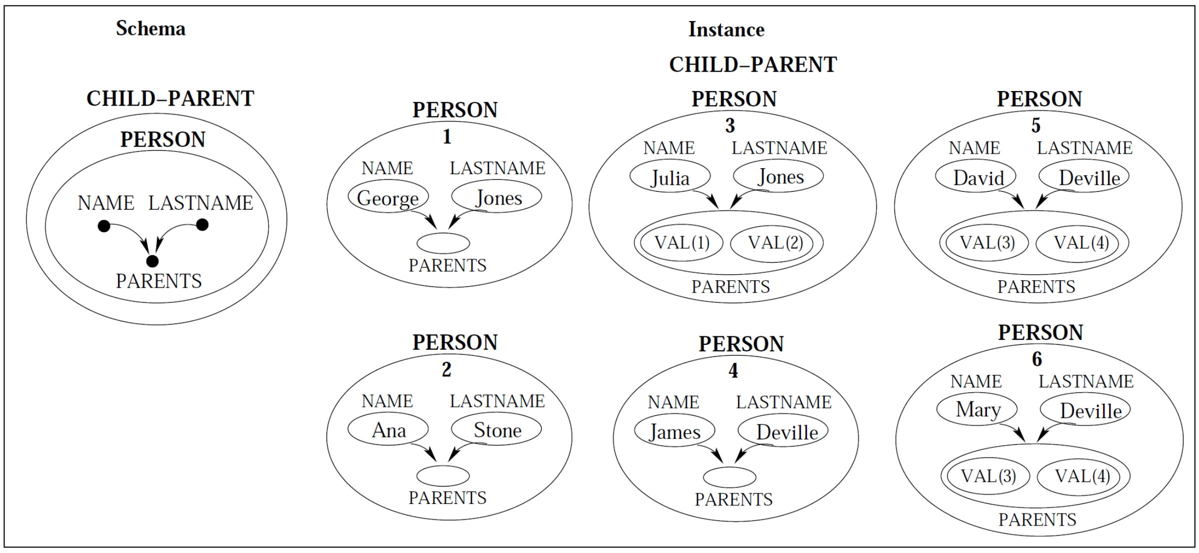
| *I (NL)* | |
|---|---|
| *1* | *val (1)* |
| 10 | (1,7) |
| 11 | (2,8) |
| 12 | (3,7) |
| 13 | (4,9) |
| 14 | (5,9) |
| 15 | (6,9) |

| *I (PP)* | |
|---|---|
| *1* | *val (1)* |
| 16 | (12,10) |
| 17 | (12,11) |
| 18 | (14,13) |
| 19 | (14,12) |
| 20 | (15,13) |
| 21 | (15,12) |

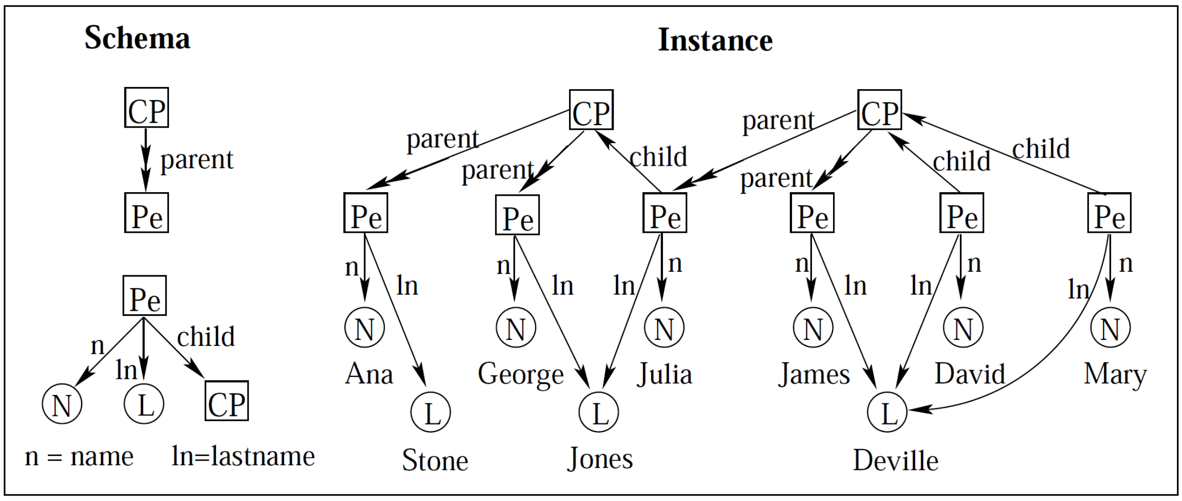
***Logical Data Model (LDM)***. The schema (on the left) uses two basic type nodes for representing data values (N and L), and two product type nodes (NL and PP) to establish relations among data values in a relational style. The instance (on the right) is a collection of tables, one for each node of the schema. Internal nodes use pointers (names) to make reference to basic and set data values defined by other nodes [Kuper & Vardi, 1984, 1993].



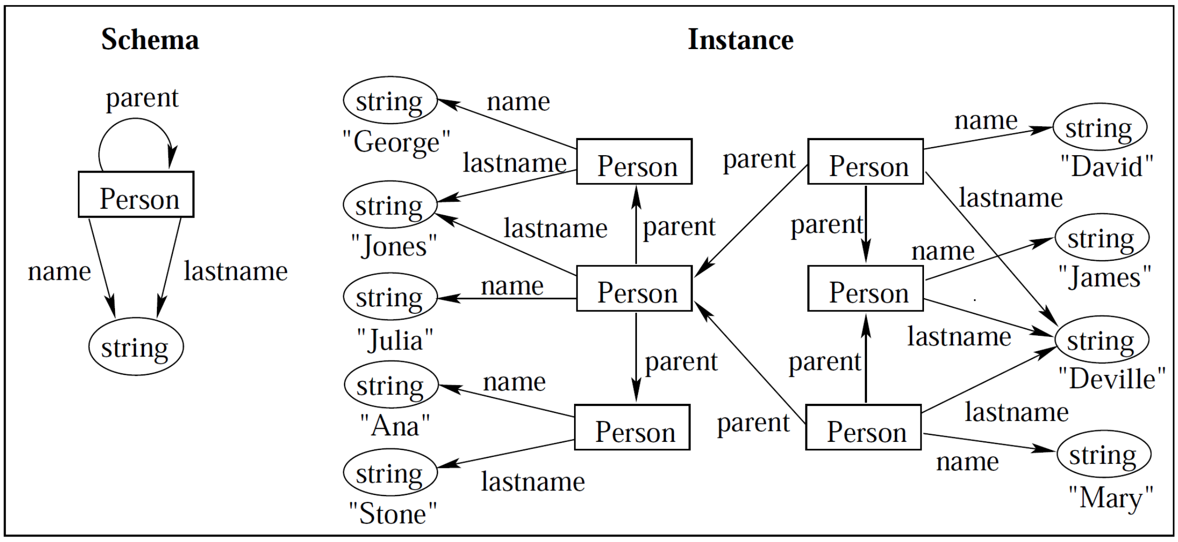
***Hypernode Mode (HyM)***. The schema (left) defines a person as a complex object with the properties name and last name of type string, and parent of type person (recursively defined). The instance (on the right) shows the relations in the genealogy among different instances of person [Levene & Poulovassilis, 1990; Poulovassilis & Levene, 1994; Levene & Loizou, 1995].



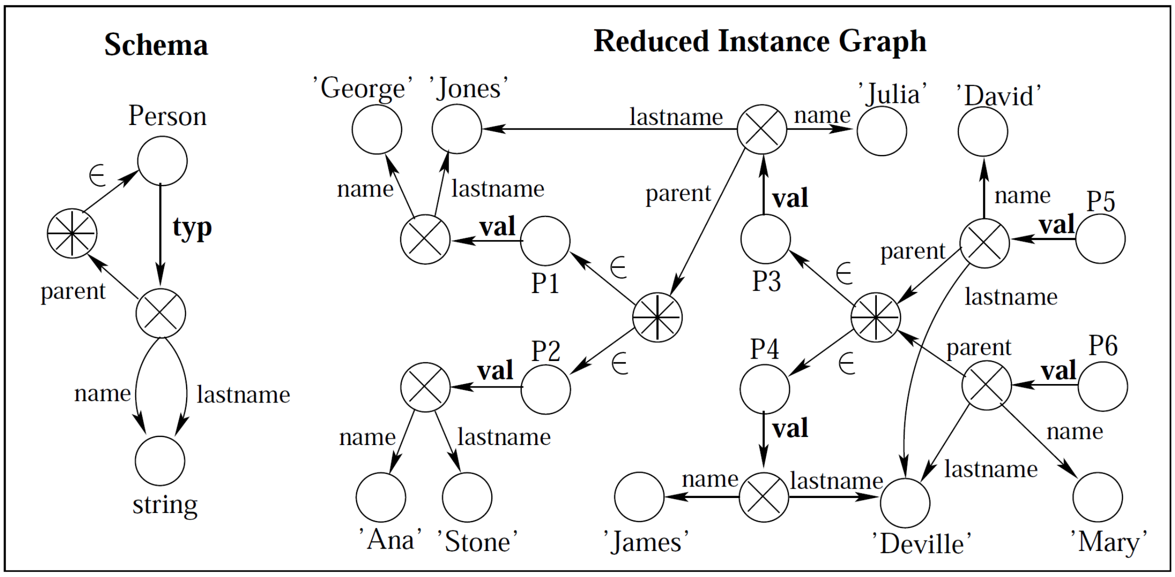
***GROOVY***. At the schema level (left), we model an object PERSON as a hypergraph that relates the attributes NAME, LASTNAME and PARENTS. Note the value functional dependency (VDF) NAME, LASTNAME → PARENTS logically represented by the directed hyperedge ({NAME, LASTNAME} {PARENTS}). This VFD asserts that NAME and LASTNAME uniquely determine the set of PARENTS [Levene & Poulovassilis, 1991].



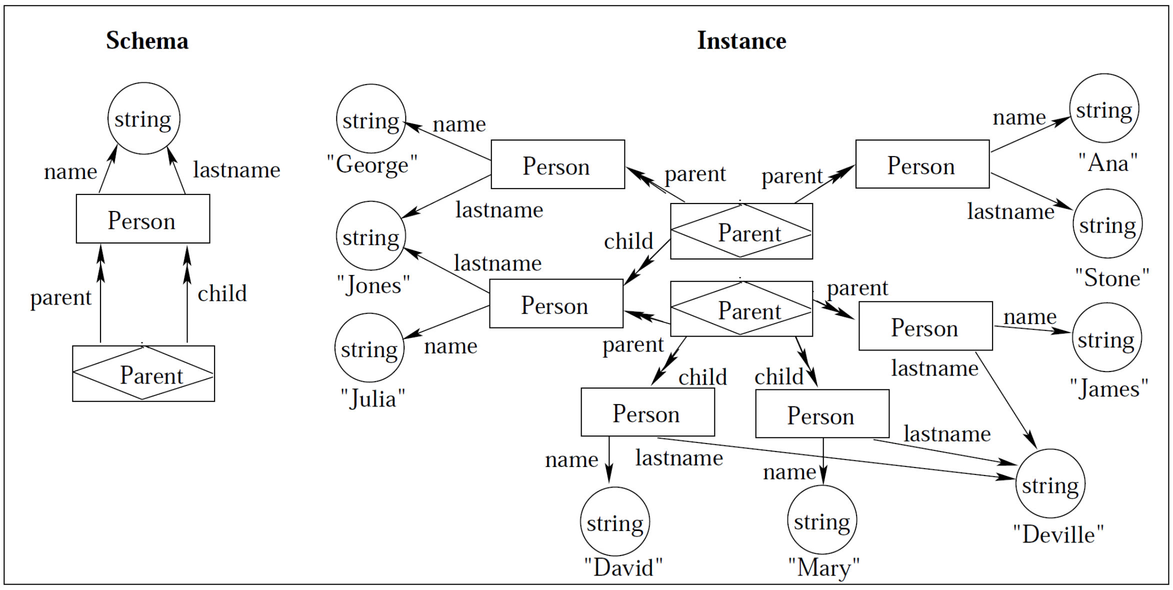
***GOOD***. In the schema, we use printable nodes N and L to represent names and last names respectively, and nonprintable nodes, Pe(rson) and CP, to represent relations Name-Lastname, and Child-Parent, respectively. A double arrow indicates a nonfunctional relationship, and a simple arrow indicates a functional relationship. The instance is obtained by assigning values to printable nodes and instantiating the CP and PE nodes [Gyssens et al, 1990; Gemis & Paredaens, 1993].



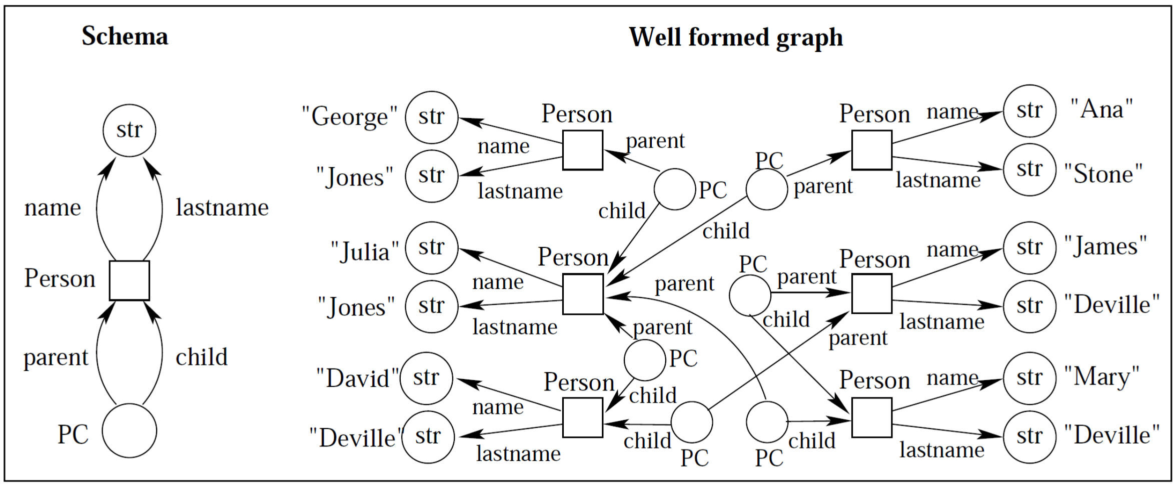
*GMOD*. In the schema, nodes represent abstract objects (Person) and labeled edges establish relations with primitive objects (properties name and last name), and other abstract objects (parent relation). For building an instance, we instantiate the schema for each person by assigning values to oval nodes [Andries et al, 1992].



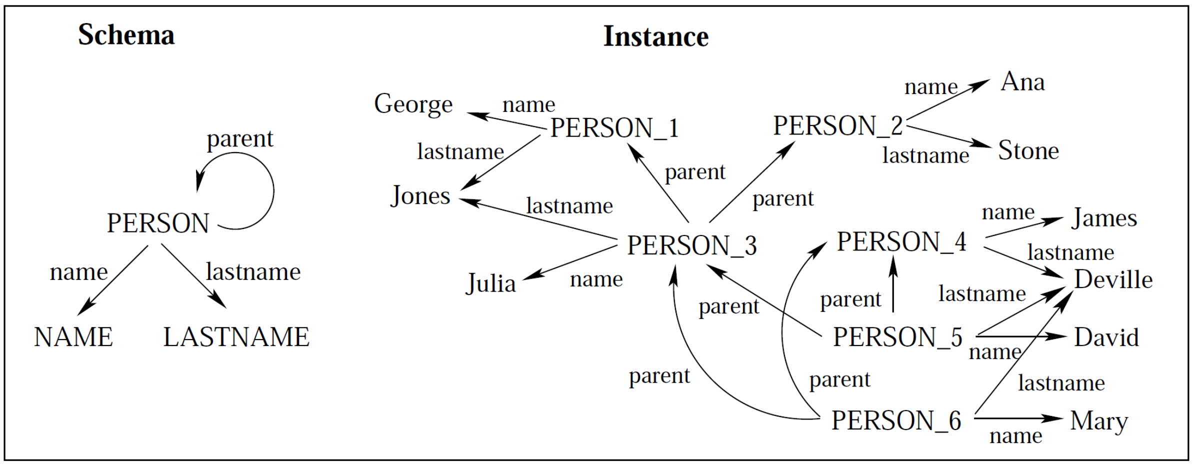
***PaMaL***. The example shows all the nodes defined in PaMaL: basic type (string), class (Person), tuple (⊗), set (⊛) nodes for the schema level, and atomic (George, Ana, etc.), instance (P1, P2, etc), tuple and set nodes for the instance level. Note the use of edges ∈ to indicate elements in a set, and the edge type to indicate the type of class Person (these edges are changed to val in the instance level) [Gemis & Paredaens, 1993].



***GOAL***. The schema presented in the example shows the use of the object node Person with properties Name and Lastname. The association node Parent and the double headed edges parent and child allow expression of the relation Person-Parent. At the instance level, we assign values to value nodes (string) and create instances for object and association nodes. Nodes with same value were merged (e.g. Deville) [Hidders & Paredaens, 1993].



***GDM***. In the schema each entity Person (object node represented as a square) has assigned the attributes name and last name (basic value nodes represented round and labeled str.). We use the composite value node PC to establish the relationship Parent-Child. Note the redundancy introduced by the node PC. The instance is built by instantiating the schema for each person [Hidders, 2001, 2002].



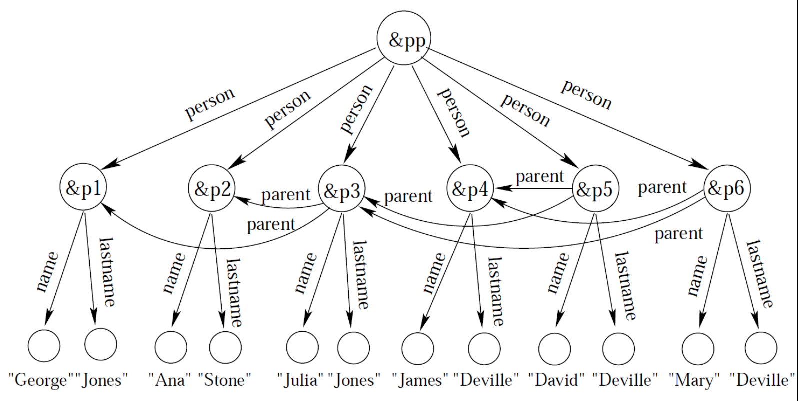
***Gram***. At the schema level we use generalized names for definition of entities and relations. At the instance level, we create instance labels (e.g. PERSON 1) to represent entities, and use the edges (defined in the schema) to express relations between data and entities [Amann & Scholl, 1992].

***Table 3. Examples of the graph models with implicit schema***

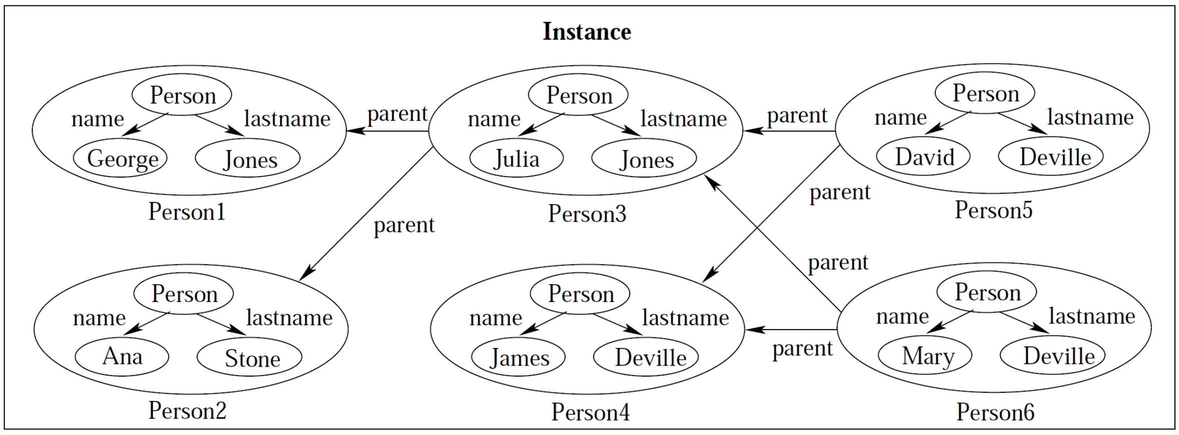
**OEM Syntax**

```
{ person : &p1 { name : "George" ,
                 lastname : "Jones" }
  person : &p2 { name : "Ana" ,
                 lastname : "Stone" }
  person : &p3 { name : "Julia" ,
                 lastname : "Jones" ,
                 parent : &p1 ,
                 parent : &p2 }
  person : &p4 { name : "James" ,
                 lastname : "Deville" }
  person : &p5 { name = "David",
                 lastname : "Deville" ,
                 parent : &p3 ,
                 parent : &p4 }
  person : &p6 { name = "Mary" ,
                 lastname : "Deville" ,
                 parent : &p3 ,
                 parent : &p4 }  }
```

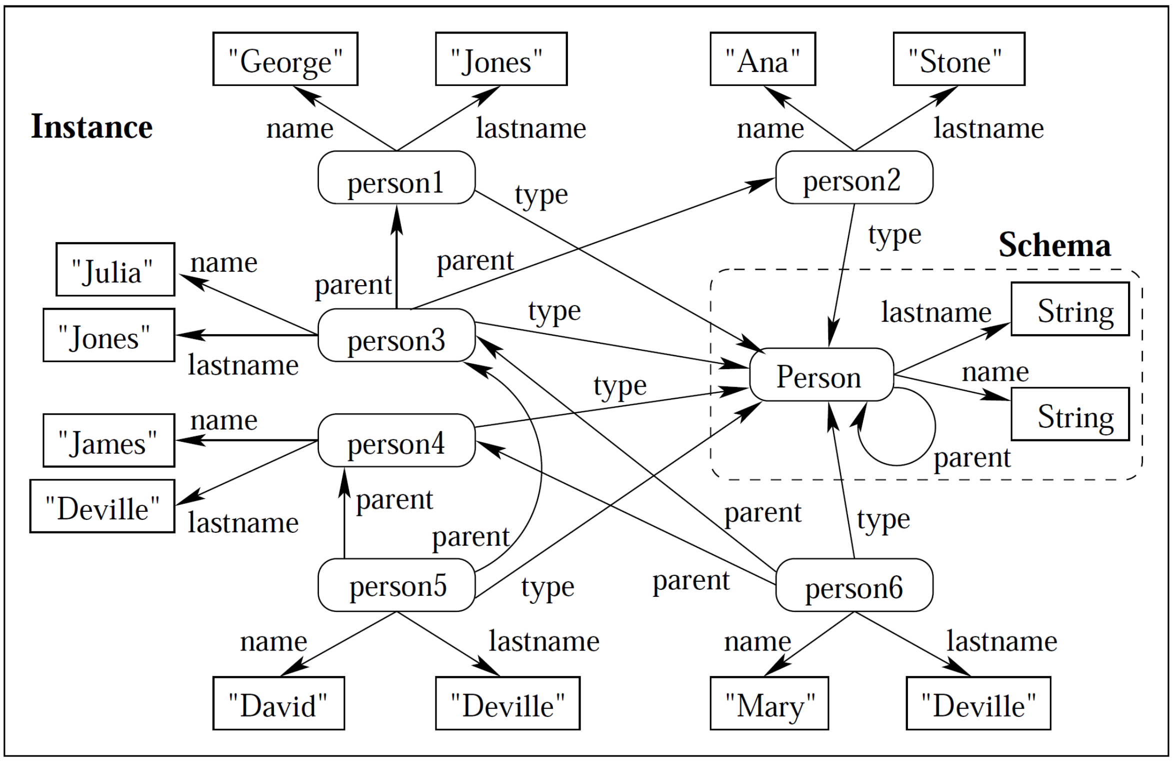
**OEM Graph**



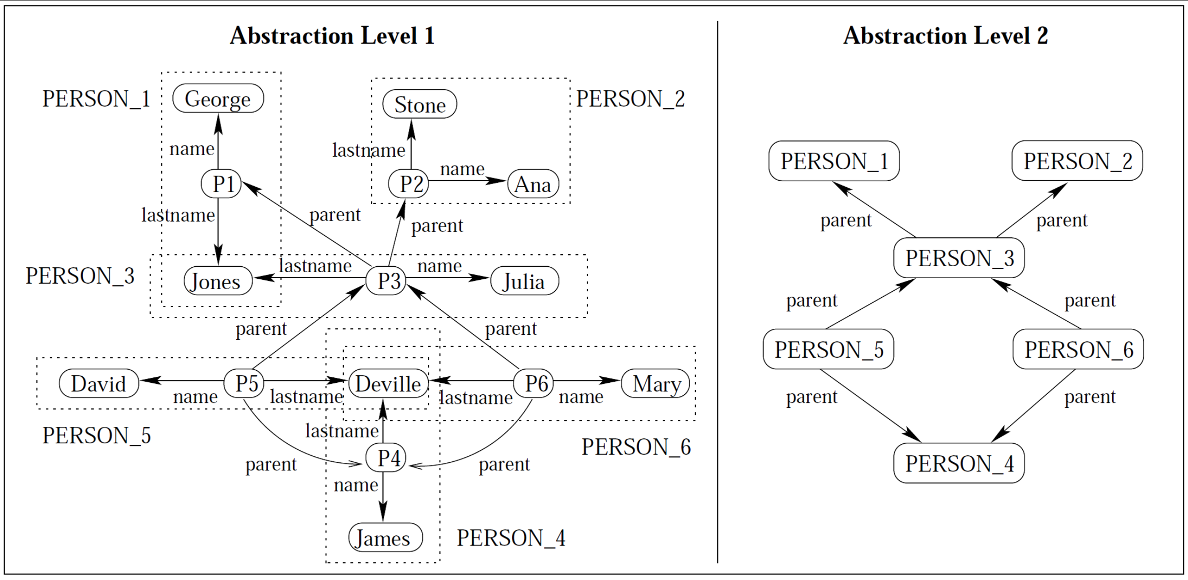
***Object Exchange Model (OEM)***. Schema and instance are mixed. The data is modeled beginning in a root node &pp, with children person nodes, each of them identified by an Object-ID (e.g. &p2). These nodes have children that contain data (name and last name) or references to other nodes (parent). Referencing permits establishing relations between distinct hierarchical levels. Note the tree structure obtained if one forgets the pointers to OIDs, a characteristic of semi structured data [Papakonstantinou et al, 1995].



***GGL***. Schema and instances are mixed. Packaged graph vertices (Person1, Person2, ...) are used to encapsulate information about the graph defining a Person. Relations among these packages are established using edges labeled with *parent* [Graves, 1993; Graves et al, 1994; Graves et al, 1995a; Graves et al, 1995b].



***RDF***. Schema and instance are mixed together. In the example, the edges labeled type disconnect the instance from the schema. The instance is built by the subgraphs obtained by instantiating the nodes of the schema, and establishing the corresponding parent edges between these subgraphs [Klyne & Carroll, 2004; Hayes & Gutierrez, 2004; Angles & Gutierrez, 2005].



***Simatic-XT***. This model does not define a schema. The relations Name-Lastname and Person-Parent are represented in two abstraction levels. In the first level (the most general), the graph contains the relations Name and Lastname to identify people (P1, ..., P6). In the second level we use the abstraction of Person, to compress the attributes Name and Lastname and represent only the relation Parent between people [Mainguenaud, 1992].

### ➢ *Advantages of Graph database models*

Graph database models are applied in areas where information about data interconnectivity or topology is more important, or as important, as the data itself. In these applications, the data and relations among the data are usually at the same level.

Introducing graphs as a modeling tool has several advantages for this type of data.

- It allows for a more natural modeling of data. Graph structures are visible to the user and they allow a natural way of handling applications data, for example, hypertext or geographic data. Graphs have the advantage of being able to keep all the information about an entity in a single node and showing related information by edges connected to it [Paredaens et al, 1995]. Graph objects (like paths and neighborhoods) may have first order citizenship; a user can define some part of the database explicitly as a graph structure [Guting, 1994], allowing encapsulation and context definition [Levene & Poulovassilis, 1990];
- Queries can refer directly to this graph structure. Associated with graphs are specific graph operations in the query language algebra, such as finding shortest paths, determining certain subgraphs, and so forth. Explicit graphs and graph operations allow users to express a query at a high level of abstraction. To some extent, this is the opposite of graph manipulation in deductive databases, where often, fairly complex rules need to be written [Guting, 1994]. It is not important to require full knowledge of the structure to express meaningful queries [Abiteboul et al, 1997]. Finally, for purposes of browsing it may be convenient to forget the schema [Buneman et al, 1996];
- For implementation, graph databases may provide special graph storage structures, and efficient graph algorithms for realizing specific operations [Guting, 1994].

Graph database models took off in the eighties and early nineties alongside object-oriented models. Their influence gradually died out with the emergence of other database models, in particular geographical, spatial, semi structured, and XML.

Recently, the need to manage information with graph-like nature especially in *RDF-databases* has reestablished the relevance of this area [Angles & Gutierrez, 2008].

## 2.7 RDF databases

**Resource Description Framework (RDF)** is the W3C recommendation for semantic annotations in the Semantic Web. RDF is a standard syntax for Semantic Web annotations and languages [Klyne & Carroll, 2004].

The design of a traditional database is guided by the discovery of regularity or uniformity. The principle of regularity is a standardization of design relying on an abstract view of the world, where exceptions to the rule are not taken into account, since they are considered as insignificant in the design of an advantageous structured schema. The popularity of relational database management systems (RDBMS) is due to their ability to support many data management problems dealt by applications. However, a priori uniformity required by relational model can lead to hardness when modeling a not static world such as Semantic Web data [Faye et al, 2012].

The primary goal of RDF is to handle ***non regular or semi-structured data***. The research community has early recognized that there is an increasing amount of data that is insufficiently structured to support traditional database techniques, but does contain a sufficiently regular structure exploitable in the formulation and execution of queries [Muys, 2007].

It is widely acknowledged that information access can benefit from the use of ontologies. For this purpose, available data has to be linked to concepts and relations in the corresponding ontology and access mechanisms have to be provided that support the integrated model consisting of ontology and data. The most common approach for linking data to ontologies is via RDF representation of available data that describes the data as instances of the corresponding ontology that is represented in terms of RDF Schema. Due to the practical relevance of data access based on RDF and RDF Schema, a lot of effort has been spent on the development of corresponding storage and retrieval infrastructures [Hertel et al, 2009].

The underlying structure of any expression in RDF is a collection of triples, each consisting of a subject, a predicate and an object. A set of such triples is called RDF graph [RDF, 2013]. This can be illustrated by a node and directed-edge diagram, in which each triple is represented as a “node-edge-node” link (hence the term "graph") (Figure 13).

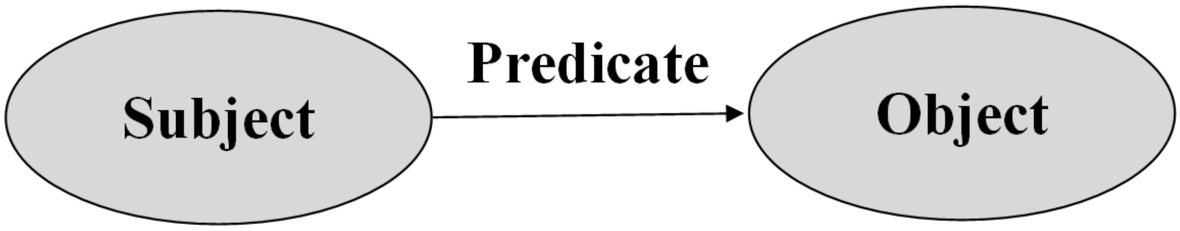


***Figure 13. RDF triple***

Each triple represents a statement of a relationship between the things denoted by the nodes that it links. It has three parts:

- Subject;
- A predicate (also called a property) that denotes a relationship;
- Object.

The direction of the edge (predicate) is significant: it always points toward the object. The nodes of RDF graph are its subjects and objects.

The assertion of RDF triple says that some relationship, indicated by the predicate, holds between the things denoted by subject and object of the triple. The assertion of RDF graph amounts to asserting all the triples in it, so the meaning of RDF graph is the conjunction (logical AND) of the statements corresponding to all the triples it contains. A formal account of the meaning of RDF graphs is given in [Hayes, 2004].

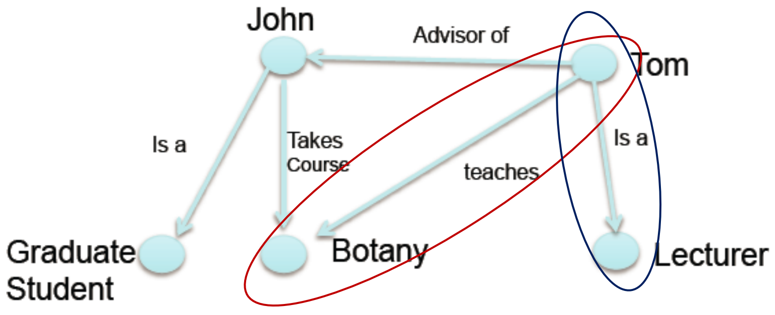
In other words, RDF provides a general method to decompose any information into pieces called triples [Briggs, 2012]:

- Each triple is of the form "Subject", "Predicate", "Object";
- Subject and Object are the names for two things in the world. Predicate is the relationship between them;
- Subject, Predicate, Object are given as URI's (stand-ins for things in the real world);
- Object can additionally be raw text.

In technical terms the RDF-triples' set form labeled directed graph, where each edge is a triple, for instance, the triples:

| **Subject** | **Predicate** | **Object** |
|---|---|---|
| <Tom> | <is a> | <Lecturer> |
| <Tom> | <teaches> | <Botany> |

define some of the elements of the next graph [Briggs, 2012]:



The research community has early recognized the natural flexibility and expressivity of triples. Indeed, triples consider both objects and relationships as first-class citizens; thus, allowing on-the-fly generation of data. The power of RDF relies on the flexibility in representing arbitrary structure without a priori schemas. Each edge in the graph is a single fact, a single statement, similar to the relationship between a single cell in a relational table and its row's primary key. RDF offers the ability to specify concepts and link them together into a graph of data [Faye et al, 2012].

## ➢ *RDF advantages*

As a storage language, RDF has several advantages [Owens, 2009]. First, it is possible to link different data sources together by adding a few additional triples specifying relationships between the concepts. This would be more difficult in the case of an RDBMS in which schema realignment or matching may be necessary. Then, RDF offers a great deal of flexibility due to the variety of the underlying graph-based model (i.e. almost any type of data can be expressed in this format with no needs for data to be present). There is no restriction on the graph size, as opposed to RDBMS field where schema must be concise. This a significant gains when the structure of the data is not well known in advance. Last, any kind of knowledge can be expressed in RDF, authorizing extraction and reuse of knowledge by various applications.

Consequently RDF offers a very useful data format, for which efficient management is needed. This becomes a hard issue for application dealing with RDF and known as ***RDF (or Triple) Stores***, due to the irregularity of the data. RDF Stores must allow the following fundamental operations on repository of RDF data: performing a query, updating, inserting (assertion), and deleting (retraction) triples [Owens, 2009]. In addition, there are issues that may require an extension of the triple-based schemas and thus are affecting the design of the database tables:

- Storing multiple ontologies in one database;
- Storing statements from multiple documents in one database.

Both points are concerning the aspect of provenance, which means keeping track of the source an RDF statement is coming from.

When storing multiple ontologies in one database it should be considered that classes, and consequently the corresponding tables, can have the same name. Therefore, either the tables have to be named with a prefix referring to the source ontology or this reference is stored in an additional attribute for every statement [Pan & Heflin, 2004].

A similar situation arises for storing multiple documents in one database. Especially, when there are contradicting statements it is important to know the source of each statement. Again, an additional attribute denoting the source document helps solving the problem [Pan & Heflin, 2004].

The concept of "***named graphs***" [Caroll et al, 2004] is including both issues. The main idea is that each document or ontology is modeled as a graph with a distinct name, mostly a URI. This name is stored as an additional attribute, thus extending RDF statements from triples to so-called quads. For the database schemas described above this means adding a fourth column to the tables and potentially storing the names of all graphs in a further table.

## ➢ *RDF disadvantages*

Different authors report different and specific RDF disadvantages. For instance, in [Costello & Jacobs, 2003] is noted that disadvantages of using the RDF format are:

- RDF uses namespaces to uniquely identify types (classes), properties, and resources. Thus, one must have a solid understanding of namespaces;
- Constrained: the RDF format constrains one on how he design his XML (i.e., one can't design his XML in any arbitrary fashion);
- Another XML vocabulary to learn: to use the RDF format one must learn the RDF vocabulary.

Other point of view we see in [Baidu, 2013]. RDF disadvantages are:

- Generic triple storage often (but not always) implies less efficient lookups (special indexes can still be built, but this moves away from schema flexibility);
- Certain data cannot easily be represented in RDF;
- Practical disadvantages with respect to (relatively) immature RDF storage systems and tools and porting over existing systems;
- High overhead for developers to get the necessary expertise to do a good job;

- Only non-standard solutions available for declaratively specifying (common types of CWA) constraints;
- The RDF triple is ontology based, always need the same schema;
- Not easy to do some complex reasoning;
- Low efficient to query data in the RDF triples, compared against RDBMS.

From our point of view, it is important to discuss the problem of ***numbering*** large RDF triple's elements (strings). Developers generally make special provisions for storing RDF resources efficiently. Indeed, rather than storing each Internationalized Resource Identifier (IRI) or literal value directly as a string, implementations usually associate a *unique numerical identifier* to each resource and store this identifier instead [Yongming et al, 2012].

There are two motivations for this strategy. First, since there is no a priori bound on the length of the IRIs or literal values that can occur in RDF graphs, it is necessary to support variable-length records when storing resources directly as strings. By storing the numerical identifiers instead, fixed-length records can be used. Second, and more importantly, RDF graphs typically contain very long IRI strings and literal values that, in addition, are frequently repeated in the same RDF graph.

Unique identifiers can be computed in two general ways [Yongming et al, 2012]:

- ***Hash-based approaches*** obtain a unique identifier by applying a hash function to the resource string, where the hash function used for IRIs may differ from the hash function used for literal values. Of course, care must be taken to deal with possible hash collisions. In the extreme, the system may reject addition of new RDF triples when a collision is detected. To translate hash values back into the corresponding IRI or literal value when answering queries, a distinguished dictionary table is constructed;
- ***Counter-based approaches*** obtain a unique identifier by simply maintaining a counter that is incremented whenever a new resource is added. To answer queries, dictionary tables that map from identifiers to resources and vice versa are constructed. Typically, these dictionary tables are stored as B-Trees for efficient retrieval. A variant on this technique that is applicable when the RDF graph is static is to first sort the resource strings in lexicographic order, and to assign the identifier n to the $n^{th}$ resource in this order. In such a case, a single dictionary table suffices to perform the mapping from identifiers to resources and vice versa [Yongming et al, 2012].

Various optimizations can be devised to further improve storage space. For example, when literal values are small enough to serve directly as unique identifiers (e.g., literal integer values), there is no need to assign unique identifiers, provided that the storage medium can distinguish between the system-generated identifiers and the small literal values. Also, it is frequent that many IRIs in an RDF graph share the same namespace prefix. By separately encoding this namespace prefix, one can further reduce the storage requirements [Yongming et al, 2012].

In other words, ***the bottleneck problem for RDF is numbering of very great amount of strings from RDF triples, sometimes up to several billion instances***.

For goal of this research we chose the second approach for solving the problem, i.e. to use counters. The new idea is that the process of numbering does not use B-Trees or any variant of traditional hashing. We use NL-addressing to assign numbers and co-ordinate access to restore string which corresponds to given number. The algorithm will be presented in Chapter 6. It has constant complexity which is important for very large datasets.

## ➢ *Storage and retrieval technologies for RDF*

The state of the art with respect to existing storage and retrieval technologies for RDF data is given in [Hertel et al, 2009] as well as in [Faye et al, 2012]. Different repositories are imaginable, e.g. main memory, files or databases.

RDF schemas and instances can be efficiently accessed and manipulated in main memory. Storing everything *in-memory* cannot be a serious method for storing extremely large volumes of data. However, they can act as useful benchmark and can be used for performing certain operations like caching data from remote sites or for performing inference. Most of the in-memory stores have efficient reasoners available and can help solve the problem of performing inference in persistent RDF stores, which otherwise can be very difficult to perform [CTS, 2012].

For persistent storage, the data can be serialized to files, but for large amounts of data the use of database management system is more reasonable. Examining currently existing RDF stores we found that they have used relational and object-relational database management systems.

Storing RDF data in a (relational) database requires an appropriate table design. There are different approaches that can be classified in:

- Generic schemas, i.e. schemas that do not depend on the ontology and run on third party databases (Jena SDB which can be coupled with almost all relational databases like MySQL, PostsgreSQL, and Oracle);
- Ontology specific schemas, for instance, the native triple stores which provide persistent storage with their own implementation of the databases (Virtuoso, Mulgara, AllegroGraph, and Garlik JXT).

Main characteristics of several known RDF triple stores and our experimental program RDFArM are presented in Table 77 of the Appendix B.

In the following we will discuss the NL-Addressing (Natural Language Addressing) as an approach to be used for organizing middle-size or large RDF triple or quadruple stores or other kind of graph data bases.

## ➢ *Storing ontology generic schemas*

### ✓ *Vertical representation*

The simplest RDF generic schema is triple store with only one table required in the database.

The table contains three columns named *Subject*, *Predicate* and *Object*, thus reflecting the triple nature of RDF statements. Indexes are added for each of the columns in order to make joins less expensive. This corresponds to the *vertical representation* for storing objects in a table [Agrawa et al, 2001].

In this case, no restructuring is required if the ontology changes. This is the greatest advantage of this schema. Adding the new classes and properties to ontology can be realized by a simple *INSERT* command in the table. On the other hand, performing a query means searching the whole database and queries involving joins become very expensive. Another aspect is that the class hierarchy cannot be modeled in this schema, what makes queries for all instances of a class rather complex [Hertel et al, 2009].

In other words, since the collections of triples are stored in one single RDF table, *the queries may be very slow to execute*. Indeed, when the number of triples scales, the RDF table may exceed main memory size. Additionally, simple statement-based queries can be satisfactorily processed by such systems, although they do not represent the most important way of querying RDF data. Nevertheless, RDF triples store scales poorly because complex queries with multiple triple patterns require many self-joins over this single large table as pointed out in [Faye et al, 2012].

The triple table approach has been used by systems like Oracle [oracledb, 2012; Chong et al, 2005], 3store [Harris & Gibbins, 2003], Redland [Beckett, 2001], RDFStore [RDFStore, 2012] and rdfDB [Guha, 2013].

## ✓ *Normalized triple store (vertical partitioning)*

The triple store can be used in its pure form [Oldakowski et al, 2005], but most existing systems add several modifications to improve performance or maintainability. A common approach, the so-called *normalized triple store*, is adding two further tables to store resource URIs and literals separately as shown in Figure 14, which requires significantly less storage space [Harris & Gibbins, 2003]. Furthermore, a hybrid of the simple and the normalized triple store can be used, allowing storing the values themselves either in the triple table or in the resources table [Jena2, 2012].

**Triples:**

| Subject | Predicate | IsLiteral | Object |
|---|---|---|---|
| *r1* | *r2* | *False* | *r3* |
| *r1* | *r4* | *True* | *l1* |
| … | … | … | … |

**Resources:**

| ID | URI |
|---|---|
| *r1* | *...#1* |
| *r2* | *...#2* |
| … | … |

**Literals:**

| ID | Value |
|---|---|
| *l1* | *Value1* |
| … | … |
| … | … |

***Figure 14. Normalized triple store***

In a further refinement, the Triples table can be *split horizontally* into several tables, each modeling an RDF(S) property. These tables need only two columns for *Subject* and *Object*. The table names implicitly contain the predicates. This schema separates the ontology schema from its instances, explicitly models class and property hierarchies and distinguishes between class-valued and literal-valued properties [Broekstra, 2005; Gabel et al, 2004].

To realize the vertical partitioning approach, the tables have to be stored by using a *column-oriented DBMS* (i.e., a DBMS designed especially for the vertically partitioned case, as opposed to a row oriented DBMS, gaining benefits of compressibility and performance), as collections of columns rather than collections of rows. The goal is to avoid reading entire row into memory from disk, like in row-oriented databases, if only a few attributes are accessed per query. Consequently, in column oriented databases only those columns relevant to a query will be read. The approach creates materialized views for frequent joins. Furthermore, the object columns of tables in their scheme can also be optionally indexed (e.g., using an unclustered B+ tree), or a second copy of the table can be created clustered on the object column. One of the primary benefits of vertical partitioning is the support for rapid subject joins. This benefit is achieved by sorting the tables via subject. The tables being sorted by subject, one has a way to use fast merge joins to reconstruct information about multiple properties for subsets of subjects.

Index-all approach is a poor way to simulate a column-store. The vertical partitioning approach offers a support for multi-valued attributes. Indeed, if a subject has more than one object value for a given property, each distinct value is listed in a successive row in the table for that property. For a given query, only the properties involved in that query need to be read and no clustering algorithm is needed to divide the triples table into two-column tables.

Inserts can be slow in vertically partitioned tables since multiple tables need to be accessed for statement about the same subject. With a larger number of properties, the triple store solution manages to outperform the vertically partitioned approach [Faye et al, 2012].

### ➢ *Storing ontology specific schemas*

#### ✓ *Horizontal representation*

Ontology specific schemas are changing when the ontology changes, i.e. when classes or properties are added or removed. The basic schema consists of one table with one column for the instance identificator (ID), one for the class name and one for each property in the ontology. Thus, one row in the table corresponds to one instance. This schema is corresponding to the *horizontal representation* [Agrawal et al, 2001] and obviously has several drawbacks:

- Large number of columns;
- High sparsity;
- Inability to handle multi-valued properties;
- The need to add columns to the table when adding new properties to the ontology,

etc.

Horizontally splitting the schema results in the so called one-table-per class schema, i.e. one table for each class in the ontology is created. A class table provides columns for all properties whose domain contains this class. This is tending to the classic entity-relationship-model in database design and benefits queries about all attributes and properties of an instance.

However, in this form the schema still lacks the ability to handle multi-valued properties, and properties that do not define an explicit domain must then be included in each table. Furthermore, adding new properties to the ontology again requires restructuring existing tables [Hertel et al, 2009].

#### ✓ *Decomposition storage model*

Another approach is vertically splitting the schema, what results in the one-table-per-property schema, also called the *decomposition storage model*.

In this schema one table for each property is created with only two columns for *Subject* and *Object*. RDF(S) properties are also stored in such tables, e.g. the table for rdf:type contains the relationships between instances and their classes.

This approach is reflecting the particular aspect of RDF that properties are not defined inside a class. However, complex queries considering many properties have to perform many joins, and queries for all instances of a class are similarly expensive as in the generic triple schema [Hertel et al, 2009].

In practice, a hybrid schema is used to benefit from advantages of combining both the table-per-class and table-per property schemas. This schema contains one table for each class, only storing there a unique ID for the specific instance. This replaces the modeling of the rdf:type property. For all other properties tables are created as described in the table-per-property approach (Figure 15) [Pan & Heflin, 2004]. Thus, changes to the ontology do not require changing existing tables, as adding a new class or property results in creating a new table in the database.

**ClassA:**

| ID |
|---|
| …#1 |
| … |

**Property1:**

| Subject | Object |
|---|---|
| …#1 | …#3 |
| … | … |

**ClassB:**

| ID |
|---|
| …#3 |
| … |

***Figure 15. RDF Hybrid schema (the table-per-property approach)***

A possible modification of this schema is separating the ontology from the instances. In this case, only instances are stored in the tables described above.

Information about the ontology schema is stored separately in four additional tables *Class*, *Property*, *SubClass* and *SubProperty* [Alexaki et al, 2001]. These tables can be further refined storing only the property ID in the Property table and the domain and range of the property in own tables Domain and Range [Broekstra, 2005]. This approach is similar to refined generic schema, where ontology is stored the same way and only storage of instances is different.

To reduce the number of tables, single-valued properties with a literal as range can be stored in the class tables [Wilkinson, 2006; Broekstra et al, 2002]. Adding new attributes would then require changing existing tables. Another variation is to store all class instances in one table called Instances. This is especially useful for ontologies where there are many classes with only few or no instances [Alexaki et al, 2001; Wilkinson, 2006; Inseok et al, 2005].

The property table technique has the drawback of generating many NULL values since, for a given cluster, not all properties will be defined for all subjects. This is due to the fact that RDF data may not be very structured. A second disadvantage of property table is that multi-valued attributes, that are furthermore frequent in RDF data, are hard to express. In a data model without a fixed schema like RDF, it's common to seek for all defined properties of a given subject, which, in the property table approach, requires scanning all tables.

In this approach, including new properties requires also adding new tables; which is clearly a limitation for applications dealing with arbitrary RDF content. Thus schema flexibility is lost and this approach limits the benefits of using RDF. Moreover, queries with triples patterns that involve multiple property tables are still expensive because they may require many union clauses and joins to combine data from several tables. This consequently complicates query translation and plan generation. In summary, property tables are rarely used due to their complexity and inability to handle multi-valued attributes [Faye et al, 2012].

This approach has been used by tools like Sesame [Sesame, 2012; Broekstra et al, 2002], Jena2 [Jena2, 2012; Wilkinson et al, 2003], RDFSuite [Alexaki et al, 2001] and 4store [Harris et al, 2009].

### ✓ *Multiple indexing frameworks*

The idea of multi-indexing is based on the fact that queries bound on property value are not necessarily the most interesting or popular type of queries encountered in real world Semantic Web applications.

Due to the triple nature of RDF data, the goal is to handle equally the following type of queries:

- Triples having the same subject;
- Triples having the same property;
- List of subjects or properties related to a given object.

For achieving this goal, these approaches maintain a set of six indices covering all possible access schemes an RDF query may require. These indexes are PSO, POS, SPO, SOP, OPS, and OSP (P stands for property, O for object and S for subject). These indices materialize all possible orders of precedence of the three RDF elements. At first sight, such a multiple-indexing would result into a combinatorial explosion for an ordinary relational table. Nevertheless, it is quite practical in the case of RDF data [Weiss et al, 2008; RDF, 2013]. The approach does not treat property attributes specially, but pays equal attention to all RDF items [Faye et al, 2012].

This approach has been used by tools like Kowari system [Wood et al, 2005], Virtuoso [Erling & Mikhailov, 2007], RDF-3X [Neumann & Weikum, 2008], Hexastore [Weiss et al, 2008], RDFCube [Matono et al, 2007], BitMat [Atre et al, 2009], BRAHMS [Janik & Kochut, 2005], RDFJoin [McGlothlin & Khan, 2009], RDFKB [McGlothlin & Khan, 2009a], TripleT [Fletcher & Beck, 2009], iStore [Tran et al, 2009], Parliament [Kolas et al, 2009].

### ✓ *Storing models for popular ontologies*

Storing models for nine popular linguistic, conceptual or mixed ontologies are outlined in Table 4. These models are similar and practically are based on the well-known file systems or relational databases (RDBMS). In the case of RDBMS, orientation is mainly toward SPARQL. The ontologies are described by high-level languages (e.g. KIF, CycL, SubL, RDF, XML), which can be interpreted and/or stored in relational structures (e.g. MySQL), ER-model (e.g. FreeBase) and others.

In general, the systems for storing ontologies and, in particular, RDF data are based on (see also [Magkanaraki et al, 2002]):

– Structures in memory (e.g. TRIPLE [Sintek & Decker, 2001]);
– Popular relational databases (e.g. ICS-FORTH RDF Suite [Alexaki et al, 2001; 2001a], Semantics Platform 2.0 of Intellidimension Inc. [ISP2.0, 2012], Ontopia Knowledge Suite [Ontopia, 2012]);
– Non-relational file systems, indexed by key B-trees using systems such as Oracle Berkeley DB (e.g. rdfDB [Dumbill, 2000], RDF Store [RDFStore, 2012], Redland [Beckett, 2001], Jena [McBride, 2001]).

***Table 4. Methods for storing data in nine ontologies***

| | **ontology name** | **developer** | **quantity of terms** | **storing models** | **integration with other ontologies** |
|---|---|---|---|---|---|
| **1** | **WordNet** [Fellbaum et al, 1998; Miller, 1995] | Princeton University | about 100 000 | files | SUMO, FrameNet |
| **2** | **Sensus** [ISI, 2012] | ISI USC | more than 70 000 | files, relational databases | subset of the WordNet |
| **3** | **Omega** [Philpot et al, 2005] | ISI USC | about 120 000 | relational databases (MySQL) | WordNet, Mikrokosmos |
| **4** | **Mikrokosmos** [Beale et al, 1996; Mikr, 2012] | CLR UNMS | more than 7 000 | relational database | WordNet, Omega |
| **5** | **OpenCyc** [OpenCyc, 2012] | Cycorp | more than 100 000 | files (CycL, SubL, RDF) | WordNet |
| **6** | **DOLCE** [Masolo et al, 2003] | LAO ICST | about 4 000 | files (KIF) | No |
| **7** | **PropBank** [Giuglea & Moschitti, 2004; Kingsbury & Palmer, 2003] | University PennState | more than 4 300 | frame files | FrameNet, VerbNet |
| **8** | **FrameNet** [Fillmore, 1976; Baker et al, 1998; FrameNet, 2012] | ISI, Berkeley, CA | about 900 frames | files (XML) | WordNet, PropBank, SUMO |
| **9** | **SUMO** [SUMO, 2012] | Teknowledge Corporation, SUO WG | more than 1 000 | SUO-KIF files | FrameNet, WordNet, EMELD |

## 2.8 Multi-layer representation of graphs

### ➢ *Names and locations in graphs*

Graph theory may be said to have its beginning in 1736 when EULER considered the (general case of the) Königsberg bridge problem:

“Is there a walk crossing each of the seven bridges of Königsberg (now Kaliningrad, Russia) exactly once?” [Euler, 1736] (Figure 16)

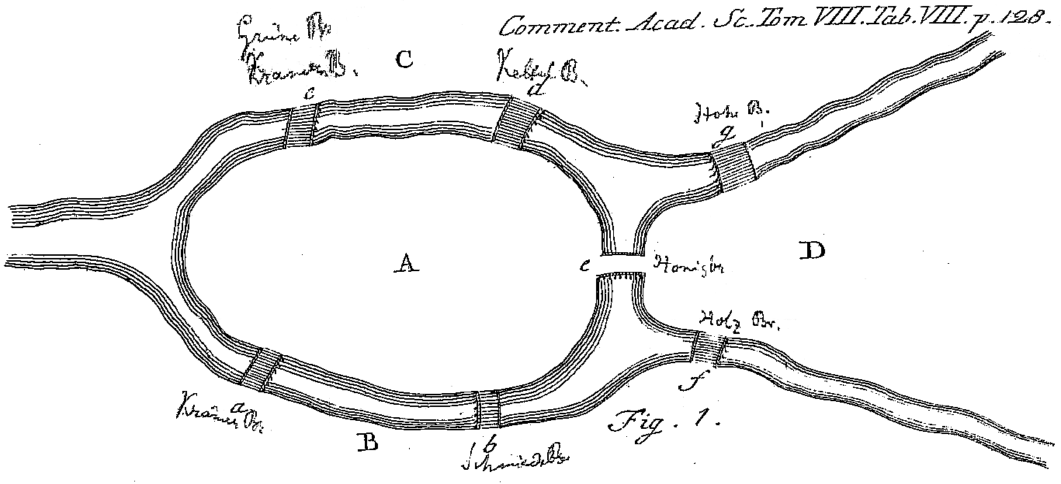


***Figure 16. Illustration of Königsberg bridge problem [Euler, 1736]***

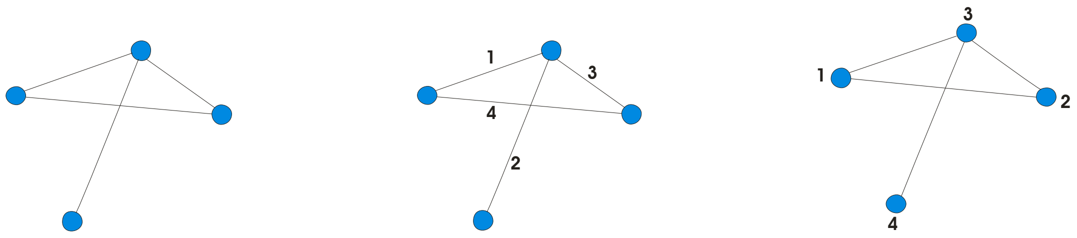
What is interesting in this schema is that every bridge has two kinds of identification - every bridge has:

- Its own name: Krämer Br. (Shopkeeper Br.), Schmiede Br. (Blacksmith Br.), Grüne Br. (Green Br.), Köttel Br. (Guts, Giblets Br.), Honig Br. (Honey Br.), Holz Br. (Wooden Br.), Hohe Br. (High Br.);
- Its own location, given by another names (addresses): a, b, c, d, e, f, g.

In this case the names are unique and different one from other. Because of this, the names of locations may seem redundant. In the general case the names may coincide, but the addresses of locations must be unique.

This way we come to the idea of “Labeled Graphs”. A graph labeling is an assignment of integers to the vertices or edges, or both, subject to certain conditions. Graph labeling was first introduced in the late 1960s. In the intervening years dozens of graph labeling techniques have been studied in over 1000 papers [Gallian, 2011].

A labeled graph *G = (V, E)* is a finite series of graph vertices ***V*** with a set of graph edges ***E*** of 2-subsets of *V*. The term "labeled graph" when used without qualification means a graph with each node labeled differently (but arbitrarily), so that all nodes are considered distinct for purposes of enumeration [Weisstein, 2013] (Figure 17).



*unlabeled graph* *edge-labeled graph* *vertex-labeled graph*

***Figure 17. Labeled graphs***

If set of names is a multi-set [Knuth, 1998], i.e. if it may contain more than one instance of the same name, than mapping the names to locations is not one-one. To make it one-one, additional qualifiers are used like "building" and "street" in two definitions below:

- **"Queen Victoria Building"**: It is a late nineteenth-century building designed by the architect George McRae in the central business district of Sydney, Australia;
- **"Queen Victoria Street"**: It is a street named after the British monarch who reigned from 1837 to 1901, located in the city of London which runs east by north from its junction with New Bridge Street and Victoria Embankment in Castle Baynard ward, along a section that divides the wards of Queenhithe and Bread Street, then lastly through the middle of Cordwainer ward, until it reaches Mansion House Street at Bank junction.

Another approach is numbering like the street numbers of houses.

At the end, the case with multi-sets of names may be resolved by the analyzing current context, for instance:

- **"King Street"**: It is a major east-west commercial thoroughfare in Toronto, Ontario, Canada. It was named after King George III, the reigning British monarch at the time the street was being built in early Toronto (then called the Town of York);
- **"King Street"**: It is a cross street in the Central Business District of Sydney, New South Wales, Australia. It stretches from King Street Wharf and Lime Street near Darling Harbour in the west, to Queens Square at St. James railway station in the east.

Depending where we are or what we talk about (Canada or Australia), saying "King Street" we will understand one or another meaning (definition) without collisions.

If set of names does not contain more than one instance of the same name, than set of locations is isomorphic to set of names, i.e. the correspondence between two sets is one-one. This means that using of one or other of these sets closely depends of the interpreter and its functionality. If the interpreter is a computer or a mathematician, the name of location (numbers, symbols, letters) are preferable. If the interpreter is an end-user (human), the natural language names (words or phrases) are preferable.

Now, one may put a question: “Is it possible, taking in account the context, one and the same string of letters to be used as both name and point to a location (address) of the definition or other information connected to it?” The positive answer is given by NL-addressing presented in this monograph.

To illustrate this, a simple example is presented in the next section.

## ➢ *Multi-layer representation of graphs*

Consider a sample graph (Figure 18) [GraphDB, 2012], which contains three nodes named (Alice, Bob, Chess) with identificators (Id.1, Id.2 and Id.3), six labeled edges connecting them (knows, knows, is_member, members, members, is_member) with identificators (Id.100,..., Id.105), and some additional features and their values.

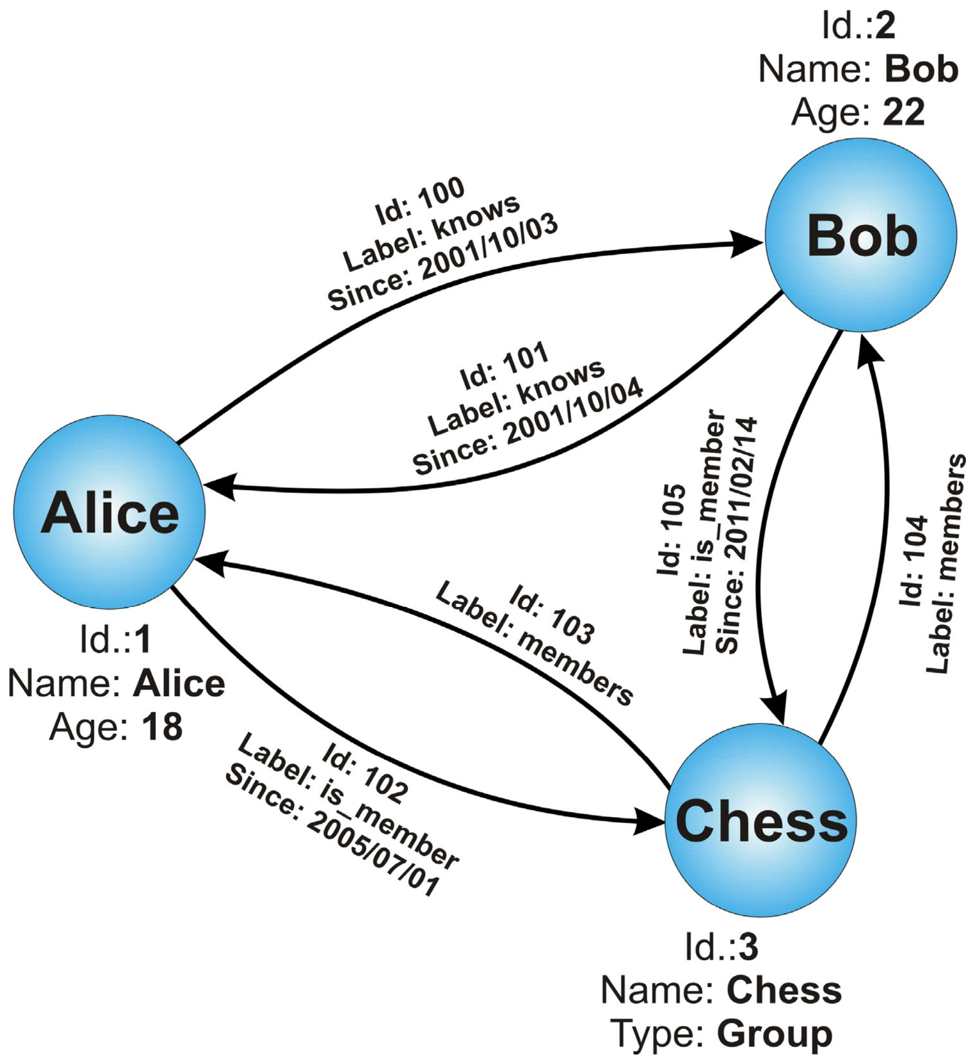


*Figure 18. A sample graph*

The set of nodes does not contain repeated members but the set of edges is a multi-set – every member is repeated two times and we need additional identification to separate different edges.

It is possible to present the information about graph in two tables – one for nodes (Table 5) and another for the edges (Table 6) [Ivanova et al, 2013b].

The names of columns and values of the corresponded cells are easy understandable. In this case we may use relational database and the additional information of locations will be the identification couples (key, column). It is well-known that in the relational databases there exits special algorithm for computing real offset in the plain file of records using the definition of the table where:

- Columns are clearly described by their names, width, type, etc.;
- Rows are identified by keys.

For Table 5 keys may be the names of the nodes or identifiers because these sets do not contain repeated elements. In this case the identifiers may be avoided.

For Table 6 the identifiers could not be avoided because the set of edges' names is multi-set, i.e. it contains repeated elements (every element is included two times). In this case, as keys we may use identifiers as single keys or couple (identifier, edge name) as complex key.

***Table 5. Description of nodes of the sample graph and index***

| index | | | information about nodes | | | | | |
|---|---|---|---|---|---|---|---|---|
| key | offset | | Node Id. | Name | from-edges | to-edges | Age | Type |
| 1 | 0 | → | 1 | Alice | 101; 103 | 100; 102 | 18 | - |
| 2 | 100 | → | 2 | Bob | 100; 104 | 101; 105 | 22 | - |
| 3 | 200 | → | 3 | Chess | 102; 105 | 103; 104 | - | Group |

***Table 6. Description of edges of the sample graph and index***

| index | | | information about edges | | | | |
|---|---|---|---|---|---|---|---|
| key | offset | | Edge Id. | Label | from node | to node | Since |
| 100 | 0 | → | 100 | knows | 1 | 2 | 2001/10/03 |
| 101 | 50 | → | 101 | knows | 2 | 1 | 2001/10/04 |
| 102 | 100 | → | 102 | is_member | 1 | 3 | 2005/07/01 |
| 103 | 150 | → | 103 | members | 3 | 1 | - |
| 104 | 200 | → | 104 | members | 3 | 2 | - |
| 105 | 250 | → | 105 | is_member | 2 | 3 | 2011/02/14 |

For both tables to reach needed information we may scan sequentially records of the file till find the right key.

Usually in relation databases, an index of keys and their offsets (locations of the rows' records) in the file is built and search is provided firstly in the index and taking the value of the offset (location) the row information is accessed directly in the main file.

For easy reading we assume that rows of the nodes' table are 100 bytes long, and rows of the edges' table are 50 bytes long.

Indexes may be of different types – B-trees, hash tables, etc. In all cases, additional resources are needed (memory to be stored and time to be processed). Note that the values of the keys are duplicated in the indexes.

At the end, the identifiers play important role for easy representing inter-relations of the graph. Because of this they could not be avoided.

If we will take in account the interrelations between nodes and edges, which define "context" of the graph, we will see that another ("***multi-layer***") representation is possible and the identifiers of nodes and edges can be avoided (Table 7 and Table 8).

Table 7 contains a multi-layer representation of the sample graph with nodes as columns and edges as layers. This is not exactly relational table because the layers may be stored in different files. If we will use the possibility for NL-addressing, the names of the columns will define locations in files of layers.

***Table 7. Multi-layer representation of the sample graph with nodes as locations.***

| | | *locations (nodes)* | | |
|---|---|---|---|---|
| | | *Alice* | *Bob* | *Chess* |
| layers | *Age* | **18** | **22** | |
| | *Type* | | | **Group** |
| | *knows; since* | **Bob; 2001/10/03** | **Alice; 2001/10/04** | |
| | *members* | | | **Alice; Bob** |
| | *is_member since* | **Chess; 2005/07/01** | **Chess; 2011/02/14** | |

To find all edges from given node we have to take node name (column) as NL-address, for instance "Bob", and read all information from different layers (rows) stored at location determined by "Bob" as NL-address.

If the information about edges is more important than we may transpose the matrix from Table 7. This will give us a multi-layer representation of the sample graph with edges as locations (Table 8). Here, the same considerations as for the Table 7 may be done.

***Table 8. Multi-layer representation of the sample graph with edges as locations.***

| | | *locations (edges)* | | | | |
|---|---|---|---|---|---|---|
| | | *knows; since* | *members* | *is_member since* | *Age* | *Type* |
| layers | *Alice* | **Bob; 2001/10/03** | | **Chess; 2005/07/01** | **18** | |
| | *Bob* | **Alice; 2001/10/04** | | **Chess; 2011/02/14** | **22** | |
| | *Chess* | | **Alice; Bob** | | | **Group** |

What is important is that NL-addressing reduces the information to be stored on the disc - only the cells with text in bold of Table 7 or Table 8 will be stored. At a rough estimate we have:

- In the “relational” case (Table 5 + Table 6): at least 18 + 30 = **48** cells for the two tables and real need of additional indexing to speed up the access. Note that empty cells in relational tables really take place on the disk (they contain space symbols);
- In the “multi-layer” case (Table 7 or Table 8): only **8** filled cells and no need of additional indexing.

In both cases the same information is stored, but in the second case the filled cells contain larger information which may be accessed by single operation and this way may speed the access.

## 2.9 Multi-domain information model (MDIM)

Below 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].

### 2.9.1 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 $\boldsymbol{\mu}_1$ be a function, which defines a biunique correspondence between elements of the set $\boldsymbol{E}_1$ $E_1$ and elements of the set $\boldsymbol{C}_1$ $C_1$of positive integer numbers, i.e.:

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

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

$$\boldsymbol{\mu}_1\ E_1 \leftrightarrow C_1$$

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 $\mu_2$ 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 $\mu_n$ 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*** $\boldsymbol{S_0}$ of level 0. It is clear that the information space $\boldsymbol{S_0 = (E_0, \mu_0, C_0)}$ is constructed in the same manner as all others:

- The machine codes (bytes) $b_i$, i=1,…,$m_0$ are considered as elements of $\boldsymbol{E_0}$;
- 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 $\mu_0$ 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 $\boldsymbol{S_0}$ 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 $\boldsymbol{S_n}$, 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 $\boldsymbol{n}$ of a basic information element. Every space address of level $\boldsymbol{m}$**,** $\boldsymbol{m < n}$, may be extended to space address of level $n$ by adding leading $n$-$m$ zero codes. Every sequence of space addresses $A_1, A_2, \ldots, A_k$, where $\boldsymbol{k}$ 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 $G = \{S_i \mid i=1,\dots,n\}$ be a set of numbered information spaces.

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

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

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

## 2.9.2 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 $S_i : E_i = \varnothing \wedge C_i = \varnothing$ , 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.

### 2.10 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 first program realizations had been done by Krassimir Markov (MDAM0, MDAM1, MDAM2, MDAM3);

The next program realizations had been done by Krassimir Markov and:

- 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***").

## ➢ *Conclusion of the Chapter 2*

*This chapter introduced the main data structures and storing technologies which further we will use to compare our results. Mainly they are graph data models as well as RDF storage and retrieval technologies.*

*Firstly we defined concepts of storage model and data model.*

*Mapping of the data models to storage models is based on program tools called “access methods”. Their main characteristics were outlined.*

*Graph models and databases were discussed more deeply and examples of different graph database models were presented. The need to manage information with graph-like nature especially in RDF-databases had reestablished the relevance of this area.*

*There is a real need of efficient tools for storing and querying knowledge using the ontologies and the related resources. In this context, the annotation of unstructured data has become a* *necessity in order to increase the efficiency of query processing. Efficient data storage and query processing that can scale to large amounts of possibly schema-less data has become an important research topic. The proposed approaches usually rely on (object-) relational database technology or on main-memory virtual machine implementations, while employing a variety of storage schemes [Faye et al, 2012].*

*In accordance with this, the analyses of RDF databases as well as of the storage and retrieval technologies for RDF structures were in the center of our attention. The analysis of the viewed tools showed that all of them use data storing models which are limited to text files, indexed data or relational databases. These approaches do not conform to the specific structures of the ontologies. This necessitates the development of new models and tools for storing ontologies which correspond to their structure.*

*Storing models for several popular ontologies and summary of main types of storing models for ontologies and, in particular, RDF data were discussed.*

*At the end of this chapter, our attention was paid to addressing and naming (labeling) in graphs with regards to introducing the Natural Language Addressing (NL-addressing) in graphs. A sample graph was analyzed to find its proper representation.*

*Taking in account the interrelations between nodes and edges, we saw that a "multi-layer" representation is possible and the identifiers of nodes and edges can be avoided.*

*Concluding, let us point on advantages and disadvantages of the multi-layer representation of graphs.*

*The main disadvantages are:*

- *The layers are sparsed;*
- *The number of locations may be very great which causes the need of corresponded number of columns in the table (in any cases hundred or thousand).*

*The main advantages are:*

- *Reducing the used resources;*
- *The NL-addressing means direct access to content of each cell. Because of this, for NL-addressing the problem of recompiling the database after updates does not exist. In addition, the multi-layer representation and natural language addressing reduce resources and avoid using of supporting indexes for information retrieval services (B-trees, hash tables, etc.);*
- *Finally, using NL-addressing, the multi-layer representation is easily understandable by humans and interpretable by the computers.*

*If we will use indexed files or relational data bases, the disadvantages are so serious that make the implementation impossible.*

*We propose to use the multi-dimensional model for organization of information. For this purpose the "Multi-Domain Infrmation Model"and its realizations were presented. The Multi-Dimensional Numbered Information Spaces are basis for context independed indexing. Because of this they may be used for storing Big Data.*