Beyond Database Schema Evolution

E. J. Yannakoudakis
Department of Informatics
Athens University of Economics and Business
76 Patission Street, Athens 10434, Greece
eyan@aueb.gr

and

I. K. Diamantis
Director, Technical High School Center
East Attica
Bari, Athens 16672, Greece
idiamant@sch.gr

ABSTRACT

This paper presents the definition of a new schema model called FDB (Framework DataBase). In modern database environments, there is a need for continuous evolution of database schemas, in order to reflect the changes that happen in the real world. The relational model permits this only through schema modification, which usually leads to time-consuming reorganization of data. On the other hand, schema evolution is supported by the object-oriented model but the commercial systems based on it present certain limitations, especially when a database is already "occupied" with data. The proposed model aims to cover the needs of dynamically evolving database environments and presents an important advantage, compared to the traditional database models: the use of FDB structures give rise to a universal logical schema, which can host any conceptual schema. In practice, this means dynamic definition and modification of entities and their corresponding attributes, without any changes in the schema structure.

Keywords: Data Bases, Data Models, Schema Evolution, Schema Modification, Dynamic Data Changes.

1. INTRODUCTION

It is well known that the relational model, the dominant database model, does not handle very well changes to the logical schema. Although the schema of a database is considered to be stable, in practice it changes, due to a variety of reasons e.g. to correct design errors, to add new features, to add new attributes or entities, and generally to accommodate changes within the actual real world environment modeled by the schema [13]. A RDBMS can accommodate such changes, but these affect heavily the applications already developed. Thus, all modules of the applications must be searched in order to locate the changes that correspond to the changes in the schema. For example, the addition of an attribute in a relation requires that all the forms and queries using this relation must be modified. Other changes to the schema are the deletion of an attribute, the modification of the degree of a relationship from 1:1 to 1:M or even M:N, the creation or dropping of a relation etc.

The need for such "evolutionary" changes necessitate the creation of dynamically evolving database environments [17, 18], which are based upon research carried out in the fields of temporal extended relational and object oriented models. In the temporal extended relational models, the structure of a database with temporal information is held within a meta-database. Changes to the structure of the database, result to changes in the semantics of the information held in the meta-database. By recording these changes we can support schema evolution. For the better manipulation of the enhanced semantics, an appropriate temporal query language is needed [10, 15, 11]. Schema versioning can also be supported using similar approaches [3, 4, 5].

In the field of object-oriented databases, schema evolution can be implemented mainly by two methods: class versioning and schema versioning [6, 7]. According to the first method, whenever a class needs modification, a new version of the class definition is created. The existing class definition is retained, allowing multiple versions of a class definition to coexist. In schema versioning each modification in the structure of a database causes the creation of a new version of the logical schema. Thus, several versions of a logical schema can be created and manipulated independently of each other. Although, most of object-oriented database systems (ENCORE, ORION, O_2, TSE, Versant etc.) support, to a certain degree, schema evolution, there appear serious limitations, especially when a database is "occupied" with data. These limitations can be summarized as follows:

- Each schema modification requires conversion (immediate or deferred) of instances.
- There is a need for extensive management of class or schema versions. Indeed, keeping track of objects created under a given class or schema version and controlling access of applications to them, appears to be a rather complex task [1, 2].
- There is no support for dynamic addition, deletion and manipulation of non-leaf classes [12].

The above limitations are caused solely from one fact: each modification to the structure of a database creates a new schema which either replaces the old or coexists with it. On the contrary, *the use of FDB structures can result to a universal logical schema, which can host any conceptual schema.*

Additionally, the proposed model presents some other important advantages compared to the relational model. As is very well known, *variable length records* and *variable length attributes* cannot be implemented directly in the relational model. Therefore, variable length attributes must be mapped into fixed length attributes or chunks of data. This design decision leads to inefficiencies, because the length chosen may either not be sufficient or it may lead to a waste of space, especially when there is significant deviation in the length of data [16]. Modern RDBMSs use special variable length data types (e.g. VARCHAR) to overcome this limitation at the expense of time-consuming operations to access these attributes. Another limitation which arises from the definition of the relational model is that *repeating groups* are not allowed. If a relation contains repeating groups, these must be separated into another relation and a reference must be kept in the first relation according to the criteria of the first normal form (1NF). The appearance of a new repeating group requires new normalization and schema reorganization, and therefore changes to existing applications. Finally, the relational model does not support directly *compound attributes* (for example, a person's or company's address); the components have to be declared individually as attributes.

In this paper, we propose FDB (Framework DataBase) as a new universal data model for dynamically evolving database environments, handling efficiently all the problems referred to above using a unified schema which caters for:

- Dynamically defined objects (relations) without schema reorganization.
- Variable length tuples (records) with a variable number of attributes (columns).
- Attributes with practically, unlimited length each.
- Repeatable and compound attributes.
- Authority records between relations.

The remainder of this paper has been divided into three sections. Section 2 first introduces the FDB model and then formally defines the structural part of it. Section 3 describes a layered implementation of the FDB model using the relational model as the underlying platform. Finally, section 4 concludes this paper and identifies directions for additional research.

2. STRUCTURAL PART OF THE FDB MODEL

2.1 Informal presentation

In this subsection, we describe and illustrate the way a conceptual schema can be modeled, using data types (structures) of the proposed model; a more formal definition of the data types of the model is presented in the next subsection.

The basic data types of the FDB model are *frame entities, frame objects, tags, subfield descriptors, fields and subfields.* They form a structure that allows easy, flexible and efficient data management and provide independence of applications from any changes to the schema. Physical or conceptual objects of the real world (we call them generally objects for the rest of the paper) are represented by *frame entities*. The characteristics (attributes) of each object are modeled by *tags*. Furthermore, compound attributes (e.g. address) can be split into single, modeled by *subfield descriptors*. Tags and subfield descriptors describe not only the domains of attributes, but also determine some important properties of them.

The aforementioned properties are essential for the utilization of the model and determine, for example, whether an attribute is optional or not, whether an attribute can contain repeatable values, whether an attribute can be referenced from others etc. More formal definitions of the properties can be found in subsection 2.3. Now, the values of the attributes are stored in *fields* and *subfields*. A *frame object* groups data (fields and subfields) belonging to a specific occurrence of an entity. As a running example for this discussion consider the E-R diagram presented in Figure 1. It models a database called *Company* and it will be used throughout the paper.

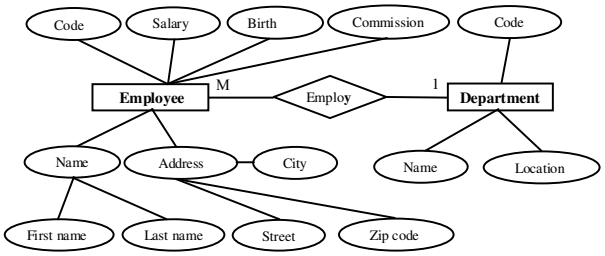

Figure 1: Conceptual schema of the *Company* database

Figure 1 shows the entities *Employee* and *Department* connected through the *Employ* relationship, assuming the type of the relationship to be 1:M. Also, Employee has the attributes *Code*, *Name*, *Address*, *Birth date*, *Salary* and *Commission*. The attributes *Name* and *Address* are compound, consisting of the simple attributes *First name*, *Last Name*, and *Street*, *City*, *Zip code* respectively. Finally, the entity Department has the attributes *Code*, *Name* and *Location*. These conceptual elements can be modeled using data types of the FDB model, as is explained in the following paragraph.

Frame entity *Employee* consists of the tags *Code*, *Name*, *Address*, *Birth date*, *Salary* and *Commission*. *Department* consists of the tags *Code*, *Name* and *Location*. Tag *Name* is composed of the subfield descriptors *First name* and *Last name*, and tag *Address* is composed of the subfield descriptors *Street*, *City* and *Zip code*. Subfield descriptor *Commission* has the *optional* property (for those employees who are not eligible for commission). The subfield descriptors *First name* and *Last name* can be repeatable (so that the corresponding subfields can store multiple values, e.g. Spanish names). The relationship between the entities will be implemented through special tags which possess the property *authority*. Therefore, a tag *Department Code* must be added to frame entity *Employee*, with its corresponding subfield values referencing the values of field, determined by tag *Code* in the frame entity *Department.*

Following the spirit of semantics [9] and object-oriented data models, the FDB model permits the explicit representation of entities through the use of identifiers. Furthermore, the characteristics (attributes) of each entity are represented by unique identifiers inside an entity. Broadly speaking, a unique identifier is assigned to each occurrence of any data type that is stored in the database. The association between identifier and occurrence is fixed and is expected to be maintainable by the DBMS. The value of the identifier is a composite value and may involve identifiers of other data type's occurrences. These composite values can be expressed using tuple, set or path constructs. Figure 2 shows how identifiers can be assigned to the occurrences of data types representing the *Company* database.

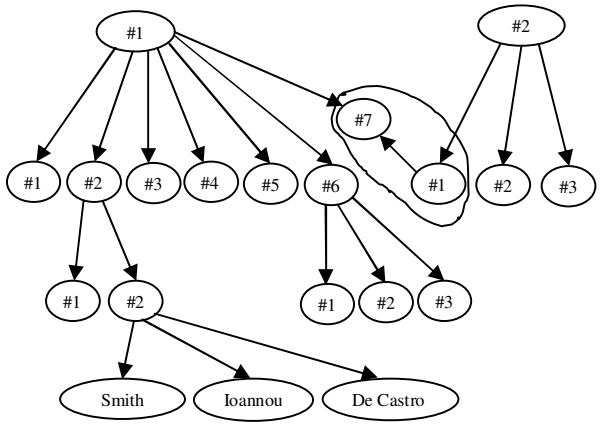

Figure 2: Structure of a database modeled with FDB

Figure 2 shows as a directed graph, that is the overall structure of *Company* database modeled using the FDB model (as root of the graph can be considered the database itself). The graph is simplified for better presentation. Top level bubbles identified by #1 and #2, correspond to the frame entities *Employee* and *Department* respectively. The tag *Code* is identified using the tuple construct as (#1,#1) and the subfield descriptor *Last name* as (#1,#2,#2). Suppose that three employees work for our example company. This means that three frame objects will hold their corresponding data. Assuming that they are identified uniquely inside the entity *Employee* by the frame object identifiers #1, #2, #3, then *Last name* "Smith" will be identified by (#1,#2,#2,#1).

2.2 Formal definitions

Now we are in position to present a formal definition of the FDB model. The presentation concerns the definitions of the data types of the model and their properties. It also concerns the integrity constraints upon the data.

To start with, we assume a collection of all atomic values and their corresponding domains: *Integer*, *Float*, *String* and *Boolean*. A special atomic value *NULL* represents an undefined value. The set ***Dom*** of atomic values will be the disjoint union of these domains. To facilitate writing queries, we assume the set ***Nam*** of named values – through them data types can be accessed directly using some name. We also assume the infinite set ***Sid*** = $\{id_1, id_2, \ldots\}$ of (simple) identifiers. Given the set ***Sid***, the tuple $(id_1, id_2, \ldots, id_n)$ and the set $\{id_1, id_2, \ldots, id_n\}$ will also be an identifier (composite). The set of all identifiers is denoted by ***Id***.

Having defined the basic sets, we proceed with the definition of each data type. A data type can be defined, either as a tuple or equivalently as a set. Data types can be virtually divided in two categories: a) data types that model conceptual elements, and b) data types that hold data, which correspond to the conceptual elements. In the first category belong: *frame entities*, *tags* and *subfield descriptors*.

- ***Frame entity*:** A data type used to determine an entity of the real world. Therefore, a frame entity represents an object or concept which has an independent existence, and can be identified uniquely. It is defined as a 2-tuple, $e = (n, m)$, where $n \in$ ***Sid***, $m \in$ ***Nam***, $h(n) = e$ and $g(m) = e$. The set of all *frame entities* is denoted ***Ent***.
- ***Tag*:** A data type used to determine a characteristic (property) of an entity. A *frame entity* has a group of *tags* associated with it. Each *tag* is uniquely identifiable inside an entity. It is defined as a 3-tuple, $t = (e, n, m)$, where $e \in$ ***Ent***, $n \in$ ***Sid***, $m \in$ ***Nam***, $h(e, n) = t$ and $g(e, m) = t$. The set of all *tags* is denoted by ***Tag***.
- ***Subfield descriptor*:** A data type used to determine a component of a compound attribute. A *tag* which corresponds to a compound attribute has a group of *subfield descriptor*s associated with it. Each *subfield descriptor* is uniquely identifiable inside a tag. It is defined as a 4-tuple, $d = (e, t, n, m)$, where $e \in$ ***Ent***, $t \in$ ***Tag***, $n \in$ ***Sid***, $m \in$ ***Nam***, $h(e, t, n) = d$ and $g(e, t, m) = d$. The set of all *subfield descriptors* is denoted by ***Subfd***.

In the second category belong *fields* and *subfields*. Their values can be considered as occurrences of the corresponding *tags* and *subfields*. The *fields* and *subfields* belonging to a specific occurrence of a *frame entity* are grouped together by *frame objects*.

- ***Field*:** A *field* is used to store data, which correspond to a specific tag inside an entity. It is defined as a 5-tuple, $f = (e, b, t, tr_i, x)$, where $e \in$ ***Ent***, $b \in$ ***Sid***, $t \in$ ***Tag***, $tr_i \in$ ***N***, $x \in$ ***Dom*** and $h(e, b, t) = f$ and $g(e, b, t, tr_i) = x$. The set of all *fields* is denoted by ***Fld***. In the above definition the identifier b is called *frame object* and identifies a group of *fields*, belonging to a specific occurrence of an entity. More formally: $\forall f \in$ ***Fld*** $\exists b \in$ ***Sid*** such that $(b, f) \in e$. The natural number tr_i identifies the i^{th} repetition of the (same) field (as we shall see in the next subsection, a field may have the property of *repetition*).
- ***Subfield*:** A *subfield* is used to store actual data, which correspond to a specific *subfield descriptor* inside a tag. It is defined as a 7-tuple, $s = (e, b, t, tr_i, d, dr_i, y)$ where $e \in$ ***Ent***, $b \in$ ***Sid***, $t \in$ ***Tag***, $d \in$ ***Subfd***, $tr_i, dr_i \in$ ***N***, $y \in$ ***Dom*** and $h(e, b, t, d) = s$ and $g(e, b, t, tr_i, d, dr_i) = y$. The set of all *subfields* is denoted by ***Sub***. As above, identifier b is a *frame object* where $\forall s \in$ ***Sub*** $\exists b \in$ ***Sid*** such that $(b, s) \in e$. The natural number dr_i identifies the i^{th} repetition of the (same) subfield (a subfield may have the property of *repetition*).

Evidently, from the above definitions each data type is actually a pair (identifier, value). As mention earlier, identifiers can be composite values and may involve other identifiers. Although there is a resemblance with other data models (a key identifies a tuple in the relational model, an OID identifies an object in the object-oriented model etc.), there is also a great difference between them and the FDB model: the structure of an identifier determines accurately the sort of a data type.

2.3 Properties

Tags and subfield descriptors are characterized by some important properties, which apply to the values of the corresponding fields and subfields (called data elements, in the rest of the paper). These properties can be assigned to tags and subfield descriptors during their definition using an appropriate Data Definition Language (DDL). In a similar manner, an attribute cannot accept NULL values in the relational model if the NOT NULL clause has been assigned to it; a foreign key references a primary key through the REFERENCES clause, etc. These properties are examined below.

- **Data length:** Data elements can be categorized as fixed or variable length. A fixed length data element has the same predefined length in all occurrences (under a specific tag or subfield descriptor), as opposed to a variable length data element where occurrences vary in length.
- **Occurrence:** Data elements can be optional or mandatory. Optional data elements may or may not appear in a frame object. Mandatory data types must appear at least once. Thus, frame objects that belong to the same entity, may not have the same structure and the same total length.
- **Repetition:** Data elements may or may not be repeatable. Repeatable data elements can appear more than once within a frame object. Non-repeatable data elements appear at most once. If a tag *t* has the property of repetition, $\exists\ f = (e, b, t, tr_i, x)$ such that $tr_i > 1$. Similarly, if a subfield descriptor *d* has the property of repetition, $\exists\ s = (e, b, t, tr_i, d, dr_i, y)$ such that $dr_i > 1$.
- **Authority:** This is a property of a tag that enables it to reference tags belonging to other entities. In this way, relationships can be implemented between entities. Specifically, if a field determined by a tag with this property references many values of another field with the same property, then a relationship of type 1:M is assumed. Additionally, if a value of the first field is referenced by many values of the second field, then the relationship type M:N is assumed.

3. AN IMPLEMENTATION OF THE FDB MODEL

The FDB model can be implemented under any database management system, either relational or object oriented. In the first case, its data types can be mapped into relations and in the second into objects. At the moment, the FDB has been fully implemented with a combination of an SQL Server and the Microsoft .NET platform. A prototype DBMS based purely on the FDB model is at an early development stage.

Let us now focus on tags, subfield descriptors, fields and subfields, which have a more complex structure. Other data types are also presented briefly. As we mention before, each field is associated with a tag, which specifies the properties of the field. A tag can be implemented as a tuple *T* (*entity, tag, name, occurrence, repetition, authority, tag_indicator1, tag_indicator2, tag_indicator3, length*) of the relation *Tag_Attributes,* where each of these attributes is defined as follows:

Attribute	Description
Entity	Frame entity identifier
Tag	Tag identifier
Name	Name of the tag
Occurrence	Indicates whether a field, associated with the specific tag, is mandatory or optional
Repetition	Indicates whether a field, associated with the specific tag, is repeatable or not
Authority	Indicates whether a tag has the corresponding property
Tag_indicator1	Provides additional information about the action that is required for a field in certain data manipulation processes
Tag_indicator2	Indicates the language of a field associated with the specific tag
Tag_indicator3	Indicates the domain of a field associated with the specific tag
Length	Indicates the length of a field associated with the specific tag

The domain of each attribute of the relation *Tag_Attributes* is defined as follows:

Dom(entity) = Integer
Dom(tag) = Integer
Dom(name) = Integer
Dom(occurrence) = {'O', 'M'}
Dom(repetition) = {'R', 'N'}
Dom(authority) = {'Y', 'N'}
Dom(tag_indicator1) = Integer
Dom(tag_indicator2) = Integer
Dom(tag_indicator3) = Integer
Dom(length) = {0, Positive integer}
Y := Corresponding property is applicable
N := Corresponding property is not applicable
O := Corresponding property is optional
M := Corresponding property is mandatory
R := Corresponding field is repeatable
0 := No data in the field
Positive integer := An integer greater than 0 denoting the length of the field.

The attributes *entity* and *name* are foreign keys, referencing the attributes *frame_entity_id* and *message_id* in the relations *Entities* and *Messages* respectively. Each tag *t* has three *indicators* associated with it. Indicators can be considered as meta-data types required in the specific implementation. The first is used to provide general information about the contents of a field, determined by the specific tag or about special manipulation processes which are required upon the contents. The second is used to determine the language of a field in multilingual database environments. Finally, the third is used to determine the domain of a field. Attributes *tag_indicator1, tag_indicator2* and *tag_indicator3* are foreign keys, referencing the primary keys of relations *Indicator1_Data*, *Indicator2_Data* and *Indicator3_Data* respectively. If a field or subfield is coded, the relations *Coded_Tags* and *Coded_Subfields* are used to store all corresponding values.

In the same way, the subfield descriptors are tuples of the relation *Subfield_Attributes*(*entity, tag, subfield, name, occurrence, repetition, subfield_indicator, length*). Each subfield descriptor has associated with it one indicator, providing information about the domain of the subfield, which is determined by the specific subfield descriptor. The other properties of a subfield (general information, special processes, and language) are inherited and controlled by the use of the first two indicators of the tag it belongs to. Application data is stored under *fields* and *subfields.* A field can be implemented as a tuple *F*(*entity, frame_object, tag, repetition, chunk, data*) of the relation *Field_Data,* where each of these attributes is defined as follows:

Attribute	Description
Entity	Frame entity identifier
Frame_object	Frame object identifier
Tag	Tag identifier
Repetition	Positive integer, indicating the specific repetition of a field
Chunk	Positive integer, identifying a chunk in a sequence of chunks used to map variable length data into fixed length slots
Data	Data of a chunk

The domains of the above attributes are defined as follows:

Dom(entity) = Integer
Dom(frame_object) = Integer
Dom(tag) = Integer
Dom(repetition) = Integer greater than 0
Dom(chunk) = Integer greater than 0
Dom(data)=String of n characters (n is equal to the attribute *length* of the relation *Tag_Attributes*)

The attributes *entity, frame_object* and *tag* are foreign keys referencing the attributes *frame_entity_id*, *frame_object_number* and *tag* of the relations *Entities, Catalogue* and *Tag_Attributes* respectively. The attributes *repetition* and *chunk* form separate sequences indicating the repetition and the chunk number of a variable length field. The relation *Subfield_Data* has a similar structure.

According to the current implementation of FDB, all data stored in fields and subfields are represented as alphanumeric characters. This is essential because the data types of the relation's attributes must be predefined. Nevertheless, the information about the real type of data is retained within *indicator3* and *subfield_indicator*, making the transformation a straightforward process, with the aid of data conversion functions [8]. Another approach that can be suggested is the use of different relations for each data type. According to this, data is stored without transformation of the attributes of the corresponding type (e.g. INTEGER, FLOAT, DATE etc.). Nevertheless the complexity added to the original structure makes this approach less attractive.

It is obvious that these limitations are imposed by the use of current RDBMS technology and will not apply in the case of an independent new DBMS based on the proposed FDB model. With an FDB-based DBMS the proper domain can be defined directly for each field or subfield.

Binary data are stored in fields where *tag_indicator1* is set properly to indicate the existence of the corresponding data. Here, the contents of the field are referencing a tuple of the relation *Bit_Data*(*entity, tag, format_type, byte_size, bit_image*). The structure of the relation *Bit_Data* is suitable for most utilities used by a DBMS to handle binary data [14].

As it was stated earlier, the relationships between entities are implemented through special tags, which own the *authority* property. An *authority tag* may be referenced by tags (with a similar property) belonging to other entities. These references are kept in the relation *Authority_Links*(*from_entity, auth_tag1, to_entity, auth_tag2*).

The logical schema proposed for an implementation of the FDB model consists of well-normalized relations, each comprising attributes with atomic values only. Each non-key attribute is fully functionally dependent on the primary key. The dependencies are presented below, where '$\rightarrow$' denotes a simple functional dependency and '$\Rightarrow$' denotes a full functional dependency, while underlined attributes form a unary set:

INDICATOR2_DATA (language_id $\rightarrow$ lang_name)
INDICATOR3_DATA (datatype_id $\rightarrow$ type)
MESSAGES (message_id $\rightarrow$ language, message)
INDICATOR1_DATA (indicator_id $\rightarrow$ ind1_value, message)
SYS_INTERFACE (sys_int_id $\rightarrow$ language, sys_message)
ENTITIES (frame_entity_id $\rightarrow$ name)
CATALOGUE (entity, frame_object_number $\Rightarrow$ frame_object_label)
TAG_ATTRIBUTES (entity, tag $\Rightarrow$ name, occurrence, repetition, authority, tag_indicator1, tag_indicator2, tag_indicator3, length)
SUBFIELD_ATTRIBUTES (entity, tag, subfield $\Rightarrow$ name, occurrence, repetition, subfield_indicator, length)
CODED_TAGS (entity, tag $\Rightarrow$ start_char, end_char, value, description)
CODED_SUBFIELDS (entity, tag, subfield $\Rightarrow$ start_char, end_char, value, description)
FIELD_DATA (entity, frame_object, tag, repetition, chunk $\Rightarrow$ data)
SUBFIELD_DATA(entity, frame_object, tag, repetition, subfield, subfield_repetition, chunk $\Rightarrow$ data)
AUTHORITY_LINKS (from_entity, auth_tag1 $\Rightarrow$ to_entity, auth_tag2)
BIT_DATA (entity, tag $\Rightarrow$ format_type, byte_size, bit_image)

Therefore, the relations are all in first and second normal form (2NF). They are also in the third normal form (3NF), because there are not transitional dependencies between the attributes. Finally, relations are in the fourth normal (4NF) because they do not contain multi-valued dependencies (a detailed proof of these declarations is out of scope of this paper).

4. CONCLUSIONS AND FURTHER RESEARCH

In this paper we presented the definition of a novel data model for dynamically evolving database environments. The main contribution of our work is to provide a data model capable of handling changes to the underlying schemata (both conceptual and logical). It is clear that the proposed data model offers some significant advantages compared to other data models. These advantages can be summarized as follows:

- Storage and manipulation of totally variable length data.
- Definition of compound attributes.
- Dynamic definition of relations with variable number of variable length attributes.
- Ability to host any changes to schemata (in the relational sense), by modifying stored values rather than by changing structures and therefore eliminating the need for reorganization at the conceptual or logical levels.

The FDB model can be easily implemented under any DBMS, as was demonstrated in the paper. However, our major objective is to implement the proposed data model as an independent database management system, which will handle directly its data types with built-in utilities as well as its own query language. Other directions for further research include the definition of an appropriate DDL and the definition of a query language.

REFERENCES

[1] **Bjornerstedt A. and Hulten C.** Version control in an object-oriented architecture. In Kim, W. and Lochovsky F. (editors), Object-oriented concepts, applications and databases, Addison-Wesley, 1989

[2] **Bertino E., Martino L.** Object-Oriented Database Systems, Concepts and Architectures, Addison Wesley, 1993

[3] **Castro D., Grandi F., Scalas M.R.** Schema Versioning for Multitemporal Relational Databases. Information Systems 22(5), pages 249-290, 1997

[4] **Grandi F., Mandreoli F., Scalas M. R.** A Generalized Modeling Framework for Schema Versioning Support, Proc. ADC 2000, Australian Database Conference, Canberra, Australia, 2000

[5] **Grandi F.** A Relational Multi-Schema Data Model and Query Language for full Support of Schema Versioning. In Proc. SEBD 2002 Nat. Conf. on Advanced Database Systems, Italy, 2002

[6] **Kim W., Chou H. T.** Versions of Schema for Object-Oriented Databases. In Proc. of 14th VLDB Conf., pages 148-159, California, 1988

[7] **Lautemann S. E.** Schema Versions in Object-Oriented Database Systems. In Proc. of 5th DASFAA Conf., pages 323-332, 1997

[8] MSDN, Books Online SQL Server 2000: Binary data types, Microsoft Corporation, 2000

[9] **Peckham J., Maryanski F.** Semantic data models. ACM Computing Surveys, 20 pages 153-190, 1988

[10] **Roddick J. F.** SQL/SE – A Query Language Extension for Databases Supporting Schema Evolution. SIGMOD Rec. 21(3), pages 10-16, 1992

[11] **Roddick J. F.** A Survey of Schema Versioning Issues for Database Systems. Information and Software Technology, 37(7), pages 383-393, 1996.

[12] **Rashid A., Sawyer P.** Evaluation for Evolution: How Well Commercial Systems Do? ECOOP 99 International workshop on Object-Oriented Databases, pages 13-24, 1999

[13] **Sjøberg D.** Quantifying Schema Evolution. Information and Software Technology, 35(1), pages 35-44, 1993

[14] SYBASE SQL Server Rel. 10.0, Transact – SQL User's Guide, Appendix B-7, 1992

[15] **Snodgrass R. T., Ahn I., Ariav G., Batory D. S., Clifford J., Dyreson C. E., Elmasri R., Grandi F., Jensen C. S., Kafer W., Kline N., Kulkanri K., Leung C. Y. T., Lorentzos N., Roddick J. F., Segev A., Soo M. D., Spipada S. M.** TSQL2 Language Specification. ACM SIGMOD Rec. 23(1), pages 65-86, 1994

[16] **Tsionos C.X. and Yannakoudakis E.J.** Database tuning using attribute statistics. Proc. 3rd HERMES Int. Conference on Computer Mathematics and its Applications, 1996

[17] **Yannakoudakis E. J., Tsionos C. X., Kapetis C.** A. A new Framework for dynamically evolving Database Environments, Journal Of Documentation, 55(2), pages 144-158, March 1999

[18] **Yannakoudakis E. J., Diamantis I.** K. Further improvements of the Framework for Dynamically Evolving Database Environments. Proc. of the HERCMA 2001, 5th Hellenic – European Conference on Computer Mathematics and its Applications, Sept. 2001.