Databases and Information Systems VIII
H.-M. Haav et al. (Eds.)

doi:10.3233/978-1-61499-458-9-57

Meta-Modelling In Document-Oriented Databases

Vilius OKOCKIS [a] and Linas BUKAUSKAS [a]
[a] *Faculty of Mathematics and Informatics, Vilnius University, Lithuania*

Abstract. With the appearance of massive databases the ever changing data sets are becoming increasingly important. Storage of massive data sets is always led by changes issued not only to actual data but meta-data that describes the structure. Meta-data is the foremost element to understand the information in databases when designing data storage and its change over long period of time. Thus there is no higher level system to comprehend unstructured data that falls into document-oriented databases. We present the Meta-model that documents itself and document, manages, and creates by using generic or universal meta-modelling constructs. The introduced method allows a flexible document structure that evolves itself over the time, reduces model creation time on the application level. We implement a novel meta-model prototype where the complexity of a model growth is proportional to its expansion scope.

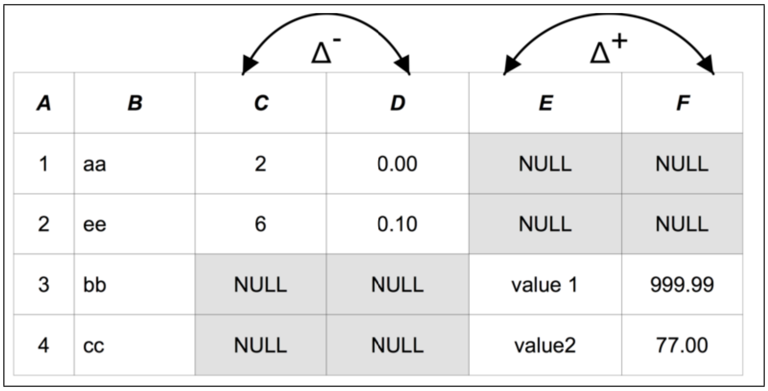
Keywords. meta-modelling, meta documents, document-oriented databases

Introduction

Vast amount of technological innovations has a significant impact on the amount of information. Notion of big data introduces detection and storage of ever changing data. Thus, large quantities of data makes it harder to maintain changes of meta data. The general practice of database design and creation is one of the approaches to organise and maintain information. Databases must conform to the requirement specification of both specified users and potential future users [1,2]. This involves requirement specification gathering phase very important and non ambiguous. Relational database model require specific notion of the domain and all changes in schema are very limited during later usage of data.

The natural question for the developer when designing changes in schema or schema that is capable of changes is: *How to store this structure in database as well as previous one?* There are several ways to deal with this problem. Denormalization and Normalization. If we use denormalization (see Figure 1), old records and meta data will have Δ^{+} attributes and new records will have Δ^{-} elements attributes which are just empty values, though they still hold data *NULL* reserved space. If we would have millions of data records, then the overhead of empty values would be huge. If we would use normalization, the complexity of databases would grow exponentially, which furthermore would influence application level complexity.

The key prerequisite is to enable users to understand information stored in databases and its evolution over the time. In most cases the user works with relational database


Δ^-
Δ^+

A	B	C	D	E	F
1	aa	2	0.00	NULL	NULL
2	ee	6	0.10	NULL	NULL
3	bb	NULL	NULL	value 1	999.99
4	cc	NULL	NULL	value2	77.00

Figure 1. Relational table with change

schema that is designed for particular tasks in mind and is not flexible enough to cope with changes [3,4,5]. Changes in highly structured and predefined data world is possible by snapshoting data sets and continuing to support backward compatible views to previous schema. Another important requirement is a continuous integration and evolution of multi-version systems in SaaS (Sofware as a service) setting [6] by maintaining intermediate schema. Such a stringent requirement brings forward described resources and information map or schema in an agreed form [7]. Such a map usually is reflected in database schema. A schema is a structure of meta-data describing how data, for example, data instances can be stored, accessed, and interpreted by users and applications [8]. Meta-data is a fundamental ground in facilitating effective resource discovery, access, and sharing across ever-growing distributed digital collections [9]. Meta-data describing data is also fundamental to integrate several heterogeneous data resources. NoSQL databases do not have effective meta-data level that helps the user to understand data and relationships in databases.

In this paper we introduce a Meta-model and a key prototype to describe and maintain evolving data. The presented prototype describes itself by supporting a flexible schema management and creation of data schema procedures. We achieve such flexibility by *generic* and *universal* meta-modelling constructs that helps us gradually build up the whole data set.

The main purpose of this work is to introduce a novel meta-model for continuously evolving and highly structured data. Meta-model manages and creates document-oriented databases that are chosen with regard to flexible and dynamic data. To prove the concept of the model, the program prototype was developed. It achieved partial system creation using meta-modelling aspects. The Meta-model creation demonstrated that its complexity growth is proportional to its expansion.

The structure of the paper is as follows. Section 1 presents related work that touch the approaches to cope with a problem and accentuate the need of universal meta-model. Section 2 describes meta-model and architecture. Section 3 presents structure of the meta-data. The discussion of model implementation with a case example is in Section 4. Conclusions and guidelines for the future work are in Section 5.

1. Related Work

The process of creating schema design is rather complex and time consuming [10]. Our proposed model aids in continuous relational [5,4] model evolution, Data Warehouse schema design creation process. Such evolving model even if it does not pin point self-documenting meta-model creation, stores data about user actions. That implies storing meta-data, user level and meta-data evolution pattern. Anchor [11] modelling distinguishes model evolution and usage. Data Warehouse schema can be incrementally modified after initial creation, however usage of such modelling is restricted to specific type of data schema and does not go beyond Relational Model to Document-oriented Databases. Our proposed method allow us to keep track of data and meta-data versions over the course of data set modification.

An alternative idea of schema design assistant is presented in [12]. Automated schema creation application for novice database designers in a way of *tab-complete* principle is discussed in that work. Auto-completion algorithm for schema is implemented using probabilities from collected Attribute Correlation Statistics Database (ACSDb). Such ACSDb acts as an assistant that provides suggesting features for attributes that would further strengthen our implementation if known or statistically defined paths for changes of meta-data would be used. This assistant can provide semantic assistance, such as suggesting features for attributes and relations. We can agree to some degree with that if we would need to create uncomplicated database schema, however ACSDb probability information is suited for low level data only.

Joseph Fong and others [13] in their work describe a semantic meta-data to preserve database constraints for data materialisation to support the user's view of database on an *ad hoc* basis. Data schema semantics can tell about behaviour and relational functionality of the object. Thus, translating different data model semantics into semantic meta-data enables us to translate it in to entirely different data model. Semantic data consist of classes: headers, attributes, methods, constraints. Header class contains information about the object and describes the structure and relations. Attribute class represents contents of the object—values, pointers to other objects or procedures characterised in methods part. Real data are stored in separate classes. Having semantic data detached, we have a mechanism for data structuring, sharing, rules and methods, which operates on information in databases. Having said that, there is no mention if this method can describe itself. The proposed meta-model solution has a disadvantage for not allowing to have more complex inner structures and does not mention about frame meta-model expansion. Our solutions lets users with base knowledge about the model, to view and to find all required information for using the database.

Self-describing models are those, that let users with basic knowledge about the model, to view and to find all required information for using the database [1]. Self-documenting models are those, that can document database evolution during long operation time. There are two main prerequisites for integrating data, schema and meta-schema. We need to create meta-schema design and data model DML. Meta-schema is the description of time invariant and general aspects how we describe schemas [2]. Nick Roussopoulos and Leo Mark presents conceptual meta-data model, which can describe itself and document evolution over the time, but it does not have possibilities to manage more complex data structures.

Yaser Karasneh and others [8] in their work states the necessity to identify syntactic suitability and semantic matching between meta-data structures for integration of database schema. Authors proposes a general framework, which facilitates integration of local schemas to global schemas. For the integration, autonomous together with heterogenous data resources are used. Its process regards to correspondence of schemas: attribute-attributes scoring and schema-schema scoring. Such method distinguishes from other methods by doing comparisons among more than two schemas. Name standardisation and ontology dictionaries are tailored by differentiating schema names, attribute names, amount of the attributes, data types and constraints. The likelihood to find similarities between schemas, which will define the successful integration, is equal to $\frac{1}{2}$. Our proposed method differs that we keep track of the history of changes in meta-data and continuously apply schema integration procedures as they occur.

Habela [7] proposes a radically simplified, flattened meta-model structure for object DBMS. By using general methods to manipulate database meta-data instead of a big set narrowly specialised operators defined by ODMG standard. Flattening enables the creation of more universal operators on meta-data in a way of simplifying its use by database designers and developers. Moreover, such model gives an opportunity to expand as it is easier to extend the dictionary than to change meta-structures. Nevertheless, meta-model becomes faster to operate and easier to maintain. Even thought, meta-data is manipulated in database level and is defined for ODBMS, our model is also applicable for creation of the meta-model for document-oriented databases.

Current meta-data or meta-schema creation practices [9,14,15] in digital repositories and collections are explored by professionals or naive users. Some implementation suggest controlled vocabularies for subject access and meta-data along with their propagation of creation directives. RAND Meta-data management system (RMMS), which controls meta-data descriptional and denotational information about databases. These works discusses standardised database schema documentation, tracking and management of different database versions, storing changes of tables, schemas and data values. The goal of complex data types, derivatives, composites, arrays, matrixes, abstract data structures is not reached by other works but is fully implemented in ours.

2. Meta-Model

Model-View-Controller (MVC) is a software pattern for implementing user interfaces. There is a number of programming frameworks based on MVC. Most of such frameworks provides quick generation of initial models, controllers and views for applications. Moreover, MVC frameworks have extensive libraries which cover complex actions under high level API. Document-oriented databases (DOD) provide schema-less database design and flexible data structure. Such pattern can be applied to produce meta-data and data management framework.

We utilised rich MVC base and DOD advantages to abstract to higher level of MVC. In Figure 2 MVC model is virtualised using Meta-data definitions. Virtual Model reduces its programming. We enabled to do that by implementing generic/universal `CONTROLLER` and `VIEW` methods. As a result, application code is enforced to be reusable and easy to refactor.

Objects in our method are identified by a sequences of pairs of `id` and `Meta-key` $(mid_1, oid_1), \ldots, ((mid_n, oid_n))$ (see Figure 2). Such pair identifies id of an instance object and meta data that describes the structure. In that way using documents we describe

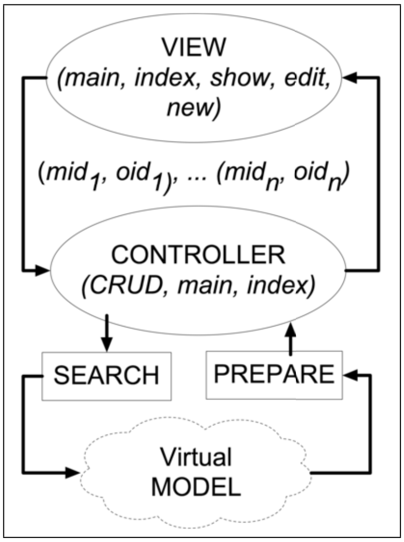

VIEW
(main, index, show, edit, new)
(mid1, oid1), ... (midn, oidn)
CONTROLLER
(CRUD, main, index)
SEARCH
PREPARE
Virtual MODEL

Figure 2. Meta-Model MVC concept

the structure of data, keep track a hierarchy of changes and allow minimisation of meta-data quantities at data instances. Also sequences enable us to `SEARCH` Meta-data of objects by pointing to the latest version of the structure, analyse it in case changes must be done, find exact data objects in database, and `PREPARE` the data for use as a global data view.

3. Meta-Data Structure

Document-oriented databases are perfectly tolerant of incomplete data. Each document has own meta-data and data attached. A collection of same type documents might have different structure that varies over the time. Thus data is hard to push to normal-form and require from the developer special application level development. Unlike DOD, relational databases hide all the complexity by having high level SQL query language that produces data-sets and collection of records according predefined rules. The key concept of DOD regards to the term of document, which places meta-data to the lower record level. Whereas relational databases place meta-data in higher table level. Document-oriented databases works with records of varied structure, whereas relational model has stringent requirement that all records are of the same structure. In that case changing meta-data in relational databases causes global table changes applied to all records. On the other hand DOD guarantees each and every record change with superfluous meta data. Document-oriented databases usually uses JSON standard for storing data and for representing data. It results to flexible record structure and therefore in terms of self-documentation it allows more freedom in model growth.

[7] Habela's proposed concept of flattened meta-model is split into three parts. As DOD is based on the term of document, we can state meta-data is stored as meta-document. Meta-documents and documents act as the very same documents and they can be built in *recurrent* manner using universal/generic methods. In Figure 3 such a self-

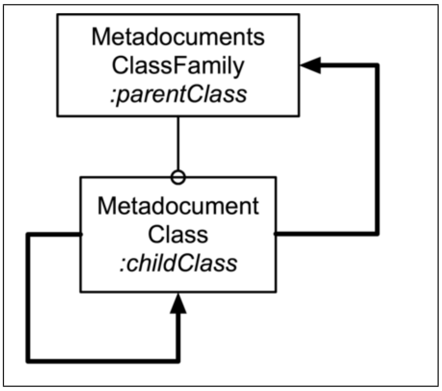


Figure 3. Generalized Meta-Model documentation structure abbreviated

documentation is proposed. We introduce *Metadocument Class* that is a part of *Metadocument ClassFamily* where both these main structural elements are recursively describing Meta-documents structure and associated data-sets.

Therefore, such Meta-data is self-documenting and stored as documents in Meta-documents collection. We propose a minimal set of Meta-Documentation Object (MDO) in Figure 4. Figure describes attributes that is a minimal set of elements. Arrows rep-

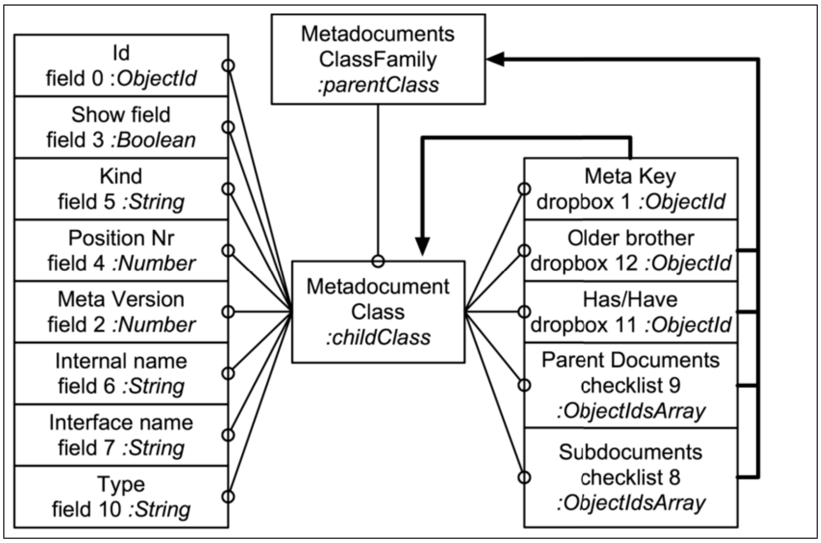


Figure 4. Meta-Model documentation structure

resent relationship to *MetaDocument Class* and *Metadocument ClassFamily*, which is needed for the model to describe itself and documents. Lines with bubbles represent MDO containment. As seen in Figure 4 left hand side, every MDO consists of unique *ID*, *Interface name*, *Kind*, *Position number*, *Type*, and optional *Show field*. Meta-Documents MDO has *ClassFamily* type, that express distinct model group. Meta-Document MDO has *Class* kind, which describes actual object's meta-information, membership of model group and enables to have varying record structures for the same model. Meta-documents

MDO embeds meta-information consisting of multiple MDO's (simple lines in Figure 4 with bubbles). Id MDO gives unique object's identification. In Figure 4 right hand side attributes describe possible relationships to keep track of changes. Meta Key MDO associates object with meta-document which describes it. Thick lines with an ending arrow seen in Figure 4 represent mentioned relations. As the model has many fields, we define meta-documents of simple *field* kind, complex *dropbox* and *checklist* kinds. Meta-documents of *dropbox* and *checklist* kind enables us to describe meta-document creation form. Type MDO represents data type or other additional interpretation and management information (Table 1). In general the proposed MDO composes treelike structure with loops and tree node notion of siblings. Such general structure can be used for different applications. The reusability of elements enable us to have minimal meta-vocabulary [7].

Table 1. Types of meta-documents.

Element type	**Description**
parentClass	root model of the parent or the parent of the current model
childClass	one version of a model
String	textual data type
Number	numerical data type
Boolean	logical data type
ObjectID	object identification type
ObjectIdsArray	inner array of document ids that assemble document from smaller components

In order to evolve meta-document, we only create new meta-document/element and assign relations. Relations are created through marking meta-document fields *child_ids* and *parent_ids*. *Dropbox* allow us to choose one of meta-documents for desirable needs such as choosing meta-key. Users can decide which data-structure to choose in application level. However, an application should have sophisticated universal and generic methods, which could understand and interpret meta-documents.

Meta-model concept is primitive and does not contain meta-information about constraints, procedures, functions. Despite this, the main idea is flexible, can adapt to the needs and focus on data documentation along with self-documentation.

4. Implementation

We structure our prototype in three tier architecture. In Figure 5 the general architecture is shown. Data store that store documents is a Document oriented database. MVC application is higher level add-on for DOD that implements handling, creation, preparation, searching functionality of meta-models. The application that implements usage of data is having a view over the REST API.

4.1. Tools

For the prototype implementation we chose Ruby on Rails, that is open source web application framework. The framework emphasises the use of well-known patterns such as MVC, convention over configuration (CoC), don't repeat yourself (DRY) and runs via the Ruby programming language. Ruby code can change aspects of its own struc-

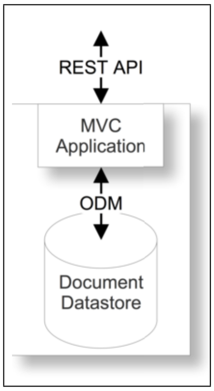


Figure 5. Implementation architecture

ture at run-time that would be pre-compiled in more rigid languages, such as class and method definitions. This sort of meta-programming can be used to write more concise code and effectively extend the language architecture. Proposed meta-model uses meta-programming extensively to code generic/universal methods, that dynamically generate objects from documented meta-data.

We implemented generic *CONTROLLER*, *VIEW*, *PREPARE* and *SEARCH* methods shown in Figure 2. Model implementation is left empty in Figure 2 as it is dynamically generated using generic/universal methods. There is no single way of implementing such methods using proposed meta-model's process pattern. We covered only well-known CRUD operations, so we exclude implementation details and provide only `SEARCH` methods pseudocode (Algorithm 1).

MongoDB was chosen for storing data. The justification for our choice of database are:

- schema-less database design and flexible data structure—allows easy and fast improvement of data model;
- document-oriented database—an object has described set of attributes. Therefore it is enough to store just one whole document. There is a possibility to distribute information across separate collections.

MongoID Object-Document-Mapper (ODM) was selected, because it provides familiar API for Ruby developers. ODM, as well as ORM [16], greatly facilitates creation of meta-model. ODM realises conversion layer between database and application.

The minimal meta-data set (see Fig. 4) was inserted into database meta-documents collection. From that point, we tested not only discussed case example (Section 4.2), but also more complex document structures like study programs, courses and subjects.

Every CRUD and other management actions employs simple meta-information `SEARCH` Algorithm 1, which traverses through meta-documents and actual documents.

The *metakey* with (or without) *_id* is a pair. A set of such pairs forms a path to a structure of specified document or to actual document. Search method looks for meta-document and its parent meta-document by using provided path from variable *param*. The method depends on the meta-key provided in actual document. This makes us aware how the record is really structured as documents in DOD stores field names at the record level. As documents can be embedded, you can not find such document without having

Algorithm 1 SEARCH Algorithm

```
Require: IN: $params$— > 1 size array where every element in odd position is meta-model ID and in every even position is document ID
Ensure: OUT: $@model$, $@document$, $@parentdocument$, $@parentmodel$
 1: $@model \leftarrow nil$ { object's meta-model}
 2: $@document \leftarrow nil$ { document's object }
 3: $@parentmodel \leftarrow nil$ { object's parent meta-model}
 4: $@parentdocument \leftarrow nil$ { document's parent object}
 5: for $element$ in $params$ do
 6:   if OddPositionInPath($element$) then
 7:     $@parentmodel \leftarrow @model$
 8:     $@model \leftarrow$ Metadocument.find($element$)
 9:   else
10:     if $@document$.nil? then
11:       $@document \leftarrow$ CastAsClass($@model$.name).find($element$)
12:     else
13:       $@parentdocument \leftarrow @document$
14:       $@document \leftarrow @document$.CastAsAttribute($@model$.name).find($element$)
15:     end if
16:   end if
17: end for
```

full path from root model. If the last pair of the path contains both elements, the method will find actual document and if not—all documents described by provided meta-key. The returning results are meta-document of the document, actual document, parent meta-document and parent document of actual document. Information about parent is needed if we want create, update or delete actual document.

Algorithm 2 *prepareAll* prepares meta information about each and every document based on their meta-data.

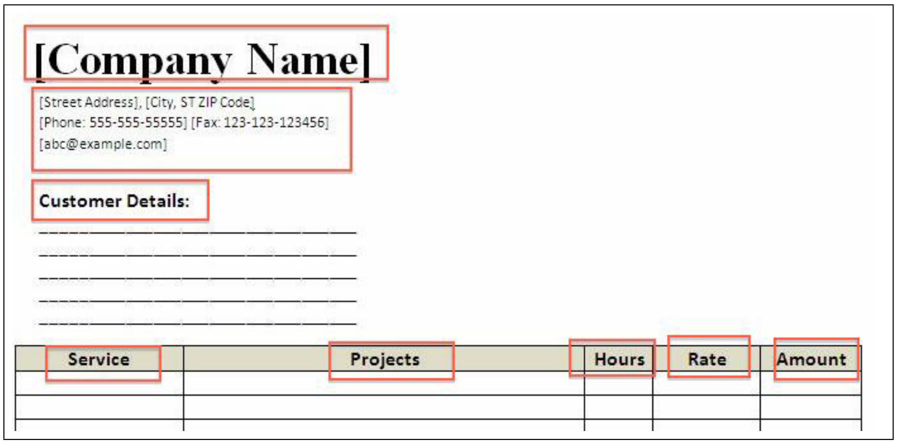
4.2. Case Study

Evolution of data used by users is natural process. It is based on documenting status and change, and is suitable for use of proposed document-referring meta-model. Consider invoice and shopping cart models to test our proposed meta-documentation. Both consists of specific quantity of attributes and have their documents, that is able to grow and evolve.

Let us consider invoice example in Figure 6. Red rectangles depict various attributes of invoice such as company details, customer details, service name, projects, hours, rate, and amount. This is very simple structure and can be easily stored using relational databases. However, companies change requirements and usually need different kind of invoice structures to be supported at the same time as well as to store old and new data associated with invoices. As the change is requested in the future (Figure 7), one would have to have different structure of invoice. Instead of customer details, we have Bill To and Ship To details. Also, instead of services—items that have description, quantity, Unit price, and amount. This illustrates how complex change might be. New meta-data and

Algorithm 2 PrepareAll Algorithm

```
Require: IN: $@documents$
Ensure: OUT: $@alldata$
1:  $@alldata \leftarrow [\ ]$
2:  $@model \leftarrow nil$ { meta model of the object }
3:  for $document$ in $@documents$ do
4:      if $@model$.id not equal to $document$.metakey then
5:          $@model \leftarrow$ Metadocument.find($document$.metakey)
6:          $@childs \leftarrow$ Metadocument.findWhere($\_id$ in $@model$.childs)
7:      end if
8:      $@data \leftarrow [\ ]$
9:      for $child$ in $@childs$ do
10:         if $document$.hasField($child$.name) then
11:             $@field \leftarrow [\ ]$
12:             $@field[0] \leftarrow child$.api_name
13:             $@field[1] \leftarrow document[child$.name]
14:             $@field[2] \leftarrow child$.posnr
15:             $@field[3] \leftarrow child$.id
16:             $@field[4] \leftarrow child$.metakey
17:             $@field[5] \leftarrow child$.name
18:             $@field[6] \leftarrow child$.showability
19:             $@data$.push($@field$)
20:         end if
21:     end for
22:     $@data$.sort($|x,y|\ (x[2] \Leftrightarrow y[2])$)
23:     $@alldata$.push($@data$)
24: end for
```



[Company Name]

[Street Address], [City, ST ZIP Code]
[Phone: 555-555-55555] [Fax: 123-123-123456]
[abc@example.com]

Customer Details:

Service	Projects	Hours	Rate	Amount

Figure 6. Invoice example at the initial state

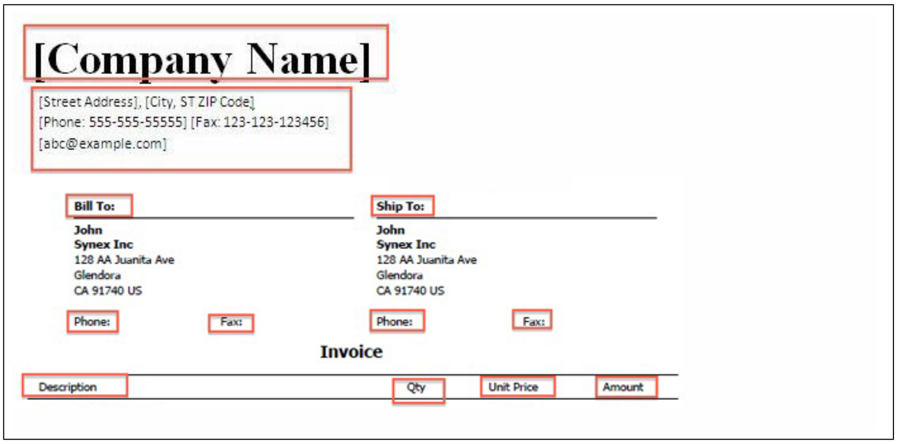


Figure 7. Invoice example with requested change

data in database as well as previous meta-data with data historically represented would be linked and would be siblings of the more general Invoice ClassFamily.

Figure 8 presents already evolved and documented both invoice and shopping cart

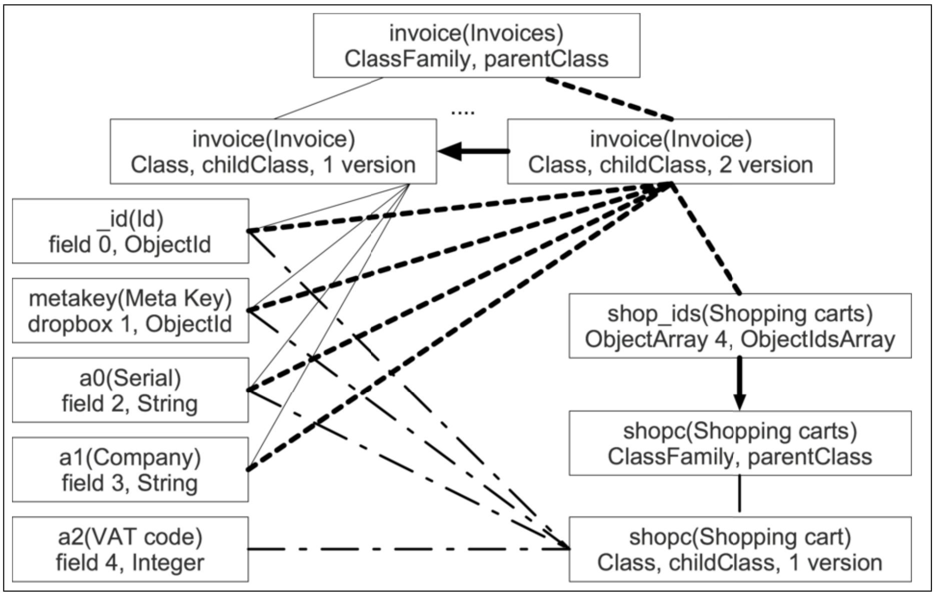


Figure 8. Invoice and shopping carts meta-documentation

models. All MDO's are denoted by rectangles. Thick lines with arrows represent relations. Three *Class* kind MDO's have different type of lines which separates embedding of MDO's. Observable relations from documentation enables us to see the story behind evolution of data structures.

Firstly, Invoices *ClassFamily* MDO were created with Invoice *Class* member as the first version of meta-documentation. Then, *Id*, *Meta Key* attributes were added to Invoice. The analogical actions are executed for shopping carts. Then, attributes *Serial* and *Com-*

pany were added to Invoice. Respectively, Shopping cart was extended with *VAT code* and *Serial* attributes. Taking this extension into account, we have the starting data structures. Although, it is clear that after some time from first Invoice structure documentation, there was a need to relate invoice and shopping carts while maintaining old records. Our proposed meta-documentation model enables us to create new Invoices member Invoice, which reuse the same attributes of 1 version Invoice and includes new MDO for one-to-many relation with shopping carts. All of such actions can be performed by using graphical user interface only.

Using created meta-documentation, we can store data of invoices and shopping carts as shown in Table 2. Physical data of new document is stored in the same collection as the old document without changing anything else in the collection. Thus two different documents in Invoices. This way of improving meta-information and adding new attributes is much simpler than in relational models where we need to change table's structure or create complex designs. A user does not need to program and change the schema of database. Database records all the history and its associated meta-data.

Table 2. Collections of invoices and shopping carts.

Invoices
{ _id:1, metaky:33, a0:"DB-2013", a1:"Amazon shop" }
{ _id:2, metaky:44, a0:"IS-2014-02", a1:"Safari shop", shopc_ids:[555] }
Shopping carts
{ _id:555, metaky:77, a0:"Groceries and NoSQL", a2:"LT" }

Also, one should notice that simple data elements have shorter inner names. This is implemented due to memory limitations. Document-oriented databases store attribute name on every field and this controls memory usage. Attribute naming convention is important if we would like t reduce storage overhead in DOD. Instead of repeating long/full attributes data is annotated with fewest number of characters. Meta-documents allow us to have a map of fields, optimise storage, and leave actual documents less touched by modification of field names. For example, in the document that describes meta-data we have relationship that attribute Company is defined as $a1$, thus every data entry instead of repeating full attribute name will have only $a1$ defined.

4.3. Implementation Issues

Proposed model is designed for document-oriented databases instead of relational object databases. In this paper we shed some light on prototype implementation and specific problems we have met while implementing meta-model:

- Implementation was limited to *one-to-many* relationships, where N object are in a separate collection. *many-to-many* relationship implementation is an analogy to *one-to-many* relationship. MongoID ODM provides *one-to-one* relationship type, though it is managed differently and now works only at meta-document creation process. Moreover, there exists storing of embedded documents inside of parent documents. However, method principles are partially identical for embedded documents. They have some peculiarities, such as adding and removing from parent document, more complex access to the field of parent document. We need to create relation type and kind (embedded or referenced) descriptors into meta-model.

It could enable model feature to become embedded and referenced at the same time dependently by described field;

- Working data types—*textual*, *numerical*, *real number*, *logical*. *Textfield*, *number-field*, *floatfield*, *checkbox*, *dropbox* are special Ruby on Rails framework elements, that have input equivalents in HTML. The latter two *checkbox* and *dropbox* fields are reserved for creation of meta-documents. *Array* type variable is implemented only partially to support addition of multiple occurrences of meta-data. The management of *Array* type acts as a set to store various models inside;
- Habela in his work [7] raises one disadvantage of model implementation in application level. It is hard and sometimes impossible to implement identical applications by using different tools. There is a possibility that Ruby on Rails framework is not suitable for the implementation of this model. Comparing implementation by different tools can be backbreaking work.
- Implemented algorithms work statically as a layer—the interpreter of meta-data is implemented to know how meta-model is structured and how to use it. Despite this limitation, creation of actual documents is not affected. This does not reflect full dynamic expansion. Changing the model through its operation is limited to interpreter level. To tackle with such limitation, interpreter should have self-learning feature.

5. Conclusion and Future Work

There are limited meta-model proposals found for document-oriented databases, but we can lean upon existing relational solutions. It should be pointed out that Document-oriented databases and relational paradigms are suited for different purposes and they have different rigidity of data structures. As Document-oriented databases provides more flexible data structures, there is a possibility to be lost in data chaos. Our proposed meta-model allows us to document data during their growth and change. Moreover, self describing and documenting data is a greatly useful feature of this model. The proposed model manages evolving dynamic meta information structures and does self documentation. It also provides generic approach of information management.

In future we will explore and analyse the remaining relation types, embedded document management, and constraints documentation. In addition to this, the idea of this work can shift to other aspects: meta-model expansion to support relational higher level languages by including different modelling principles as well as graphical modelling techniques, to change the concept of meta-model to support geographical and bitemporal aspects of data.

References

[1] Roussopoulos N., Mark L. Schema manipulation in self-describing and self-documenting data models. *International Journal of Parallel Programming*, 1985, 14(1), 1–28.

[2] Mark L., Roussopoulos N. Metadata Management. *IEEE Computer Magazine*, 1986, 19(12), 26–36.

[3] Moon H. J., Curino C., Ham M., Zaniolo C. PRIMA: archiving and querying historical data with evolving schemas. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. New York, USA: ACM, 2009. 1019–1022.

[4] Guttorm O. J., Böhlen M. H., Multitemporal Conditional Schema Evolution. *ER (Workshops)*, 2004. 441–456.

[5] Guttorm O. J., Böhlen M. H. Evolving Relations. *FMLDO*, 2000. 115-132.

[6] Yan J., Zhang B. Support Multi-version Applications in SaaS via Progressive Schema Evolution. In: *Proceedings of the 25th International Conference on Data Engineering*, Shanghai, China: IEEE, 2009. 1717-1724.

[7] Habela P. Metamodel for Object-Oriented Database Management Systems[phd thesis]. Warsaw, Poland, 2002.

[8] Karasneh Y., Ibrahim H., Othman M., Yaakob R. A Model for matching and integrating heterogeneous relational biomedical databases schemas. In: *Proceedings of the 2009 International Database Engineering & Applications Symposium*. New York, USA: ACM, 2009. 242–250.

[9] Park J. R., Tosaka Y. Metadata creation practices in digital repositories and collections: Schemata, selection criteria, and interoperability. *Journal of Technology and Libraries*, 2010, 29(3), 104–116.

[10] Arfaoui N., Akaichi J. A Data Warehouse Assistant Design System Based on Clover Model. *International Journal of Database Management Systems*, 2010, 2(2), 57–71.

[11] Ronnback L., Regardt O., Bergholtz M., Johannesson P., Wohed P., Anchor modeling Agile information modeling in evolving data environments, *Data & Knowledge Engineering*, Volume 69, Issue 12, December 2010, Pages 1229–1253.

[12] Cafarella Michael J., Alon H., Jayant M. Structured Data on the Web. *Journal of Communications of the ACM*, 2011, 54(2), 72–79.

[13] Fong J., Shiu H., Fei Yeung Y. Concurrent Data Materialization for XML-Enabled Database with Semantic Metadata. *International Journal of Software Engineering and Knowledge Engineering*, 2010, 20(3), 377-422.

[14] Cammarata S., Kameny I., Lender J., Replogle C. The RAND Metadata Management System (RMMS). A Metadata Storage Facility to Support Data Interoperability, Reuse, and Sharing. Santa Monica, *CA: Rand*, 1995. 50 p.

[15] Richardson C., ORM in dynamic languages. *Journal of Communications of the ACM - A Direct Path to Dependable*, 2009, 52(4), 48–55.

[16] O'Neil E. J. Object/relational mapping 2008: hibernate and the entity data model (edm). In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, USA: ACM, 2008. 1351-1356.