Demonstration

MM-evolver: A Multi-Model Evolution Management Tool

Demo Paper

Michal Vavrek
Department of Software Engineering, Charles University
Prague, Czech Republic
vavrekmichal@gmail.com

Irena Holubová
Department of Software Engineering, Charles University
Prague, Czech Republic
holubova@ksi.mff.cuni.cz

Stefanie Scherzinger
OTH Regensburg
Regensburg, Germany
stefanie.scherzinger@oth-regensburg.de

ABSTRACT

There is a new generation of databases specifically addressing *Big Data Variety*: *Multi-model* databases store and process structurally heterogeneous data, managing several data models in one integrated backend. Yet one of the many challenges these systems face is evolution management. In our demonstration, we present our prototype implementation of a tool called *MM-evolver*. *MM-evolver* carries out user-triggered schema modification operations over a multi-model database, and propagates them across all models. As a novelty, *MM-evolver* supports both inter- and intra-model schema modification operators. To the best of our knowledge, ours is the first tool addressing evolution management in the world of multi-model databases.

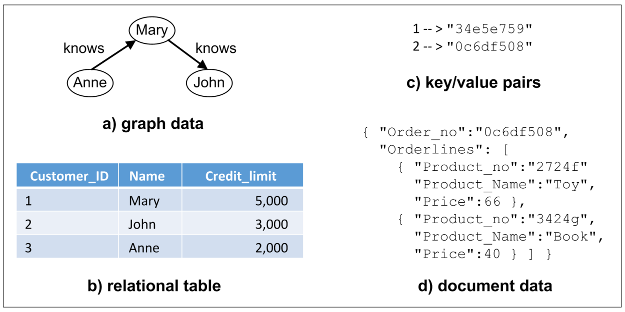

Figure 1: A multi-model data scenario [6]

1 INTRODUCTION

In recent years, the Big Data movement has broken down the borders of many technologies and approaches that have so far been acknowledged as mature and robust. One of the most challenging issues is the "*Variety*" of Big Data. Data may be present in various types and formats – structured, semi-structured, and unstructured – and produced by different sources, and hence natively have various models.

The challenge of handling variety has inspired a new generation of dedicated *multi-model databases* (MMDs), capable of storing and processing structurally different data, by supporting several data models in a single DBMS having a unified query language and API. The MMD way of solving the polyglot persistence problem offers advantages in data modeling, allowing to represent data in its most native model. This can be considered as opposite to the "*One Size Does Not Fit All*" argument [10]. However, it can be also understood as a way of re-architecting traditional database models to address new requirements [4]. Nothing shows the picture better than the Gartner Magic quadrant for operational database management systems [3], which (correctly) assumed that, by 2017, the majority of leading DBMSs will offer multiple data models in a single DBMS platform.

To illustrate the challenge of multi-model data management, consider the simple example depicted in Figure 1. There we have data with four distinct data models. Customer data is stored in a relational table – their ID, name, and credit limit. Graph data bear information about mutual relationships between the customers, i.e., who knows whom. In JSON documents, each order has an ID and a sequence of ordered items, each of which includes product number, name, and price. The fourth type of data, key/value pairs, bears a relationship between customers (or rather, their IDs) and orders (or rather, their IDs).

One of the many challenges [6] these systems are facing is evolution management. As user requirements change, the data structures evolve, and, consequently, also the respective storage strategy, queries etc. This problem is challenging even in the world of single-model databases. In simpler applications, we can rely on a skilled DB administrator, but in more complex situations it is a difficult and error-prone task. In addition, we can observe contradictory approaches to this problem in different types of DBMSs. In the world of traditional relational or XML databases, the evolution of data structures requires immediate changes in the schema. With NoSQL systems, we can (to some extent) exploit the schemalessness and ability to store data with similar, but not necessarily the same structure. Considering the existence of a schema, another complication is existence of schema-full, schema-less and even schema-mixed MMDs. Consequently, the problem of evolution management in MMDs is much more complex.

Consider again Figure 1. We may want to add a new property to one of the models (e.g., an address to JSON documents representing orders), which does not affect the other models. But, later we may decide to move this property to another model (e.g., to represent addresses in the relational model instead). Hence, we need to change data in both models. In addition, there might already exist a reference to the modified property, which then needs to be updated accordingly.

To address the indicated problems, we extend our previous research results [7, 8] for single-model systems (XML or relational) or systems with closely related models (namely aggregate-oriented NoSQL). In this demonstration, we present a tool called *MM-evolver*, which carries out user-required changes over a multi-model schema and propagates them across all sub-models. To the best of our knowledge, this is the first solution addressing the problem of evolution management in the world of multi-model databases. We see it as the first step towards a unification of evolution management across multiple models.

In the remainder of this paper, we introduce the ideas implemented in *MM-evolver* and outline our demonstration.

2 MULTI-MODEL EVOLUTION MANAGEMENT

There are two main approaches to supporting different models:

- **Complex engine** (e.g., CouchBase [2]) – The DBMS transforms all supported data types to a single core model. Its engine has to pre-process and map all operations to the core model.
- **Layer-based architecture** (e.g., Oracle Database 12c [1]) – The DBMS supports different models via different layers on top of an engine. Data are stored in the relevant model. Each data model has its own component which communicates with the engine.

We focus on the layer-based architecture, which is used in a significant portion of existing MMDs, because there is no need to introduce a generic approach for specific complex engines, since they often internally map all supported models onto a single model. Figure 2 shows two main layers of the layer-based approach inspired by the principles of the Model Driven Architecture: *model-specific* and *model-independent*[1]. For the sake of simplicity we assume that the data in the individual models have a schema. However, such a schema does not have to be explicitly defined. It can be a kind of an agreed structure, as often used in practice. The engines in the model-specific layer can thus differ also with regards to this aspect.

The MMD engine is located in the model-independent layer. It is a facade for functions of the database, such as queries, and distributes queries and commands to the respective individual models. Also, it collects data from them and creates the final result for the user.

2.1 Database Schema Evolution Language

Our aim is a general solution for schema evolution in MMDs and the following models as the most common representatives: (1) relational, (2) column, (3) graph, (4) key/value, and (5) document (i.e., JSON or XML). By the generic term *kind*, we refer to an abstract label that describes or groups related records. In the relational model, this corresponds to a table. Some MMD vendors use the terms *class* (as in OrientDB[2]) or *collection* (as in ArangoDB[3]).

First, we settle on a common set of operations which can be supported by all models. For this purpose, we extend the work from [9], where the *NoSQL Schema Evolution Language* (NoSQLSEL) covers most of the representatives, namely the aggregate-oriented NoSQL databases, i.e., document, column and key/value models. It involves three basic operations that affect all entities of a given kind to (1) *add* (introducing a new property with a specified default value), (2) *delete* (removing a property), and (3) *rename* (changing the name of a property). It further involves operations to (4) *move* (removing a property from one kind of entity and adding it to another one), and (5) *copy* (copying a property from one kind of entity to another one).

In order to avoid complex extensions of NoSQLSEL towards the missing models, we use a strategy common to most of the existing MMDs [5], i.e., a kind of a unification of the models. For example, we can treat the graph model like ArangoDB, where special edge collections bear information about edges in a graph whose nodes correspond to documents. Similarly, we can assume that entities are represented as rows in a specific table and their properties are columns of the table, where each entity has a unique identification *id*. We call this extension covering the model-specific layer the *Database Schema Evolution Language*.[4]

2.2 Multi-Model Schema Evolution Language

Having a common interface supported by all models in the model-specific layer, we can introduce the *Multi-Model Schema Evolution Language* (MMSEL) which is executed in the model-independent layer. The multi-model engine has to distinguish which models are affected by a given operation and propagate the operations to them. We can divide the operations into two separate groups:

- *Intra-model* operations (i.e., *add*) affect just one model.
- *Inter-model* operations (i.e., *copy*, *move*, *delete* and *rename*) can affect multiple models[5]. The first two can also trigger data transfer between a *source* and a *target* model; the last three can trigger changes in other models, due to references that need to be updated accordingly.

Figure 3 shows the EBNF grammar for the MMSEL syntax and Figure 6 shows an example statement. Intra-model operations, as well as inter-model operations operating within a single model are propagated by the multi-model engine to the specific target model(s) which is/are already able to ensure correct data processing.[6] When entities should be transferred between models (i.e., copied or moved), the multi-model engine gets all entities of the given kind from the source model and inserts them into the target model. In case of operation *move*, it has to delete them from the source model. It is also able to track all cross-model references. When the engine detects a change in a referenced entity, it propagates these changes to the referencing entity.

2.3 Implementation of MMSEL

The core logic of the MMSEL happens in the model independent layer. Internally, MMSEL schema modification operations are translated into a lower-level language, which we introduce next. To distinguish between the models, we introduce the *data model set* $DMS = \{column, document, key/value, graph, relational\}$ and a *model key* δ: $\delta \in DMS$. To create an abstract model of the MMD, we follow the notation from [9] which uses the term *application state* for the current state of the application space. It is a non-persistent application memory. *Database state* is the current state of the database and it represents all stored data.

We need to call specific schema evolution functions in specific models. Consequently, we introduce a modified set of functions called Multi-Model Database Programming Language (MMDPL) which extends the NoSQL Database Programming Language [9] with DMS operations (see Figure 4). The main difference is in operations for getting entities from the database and to save them in the database. Function *empty* does not modify the application space or the database. Despite the original plan of having a common set of operations, we decided to use it for the key/value model, where there is no support for operation *move* since in this model, entities do not have several properties, just a single, opaque value. Otherwise it could be implemented as *rename*. The remaining operations are extended with the DMS, but their logic remains. Rule 7 extends function $put(\delta, \kappa)$ by parameter δ to distinguish where the entity with the key κ is stored. For that purpose we introduce function $model(\kappa)$ which returns a

[1] We borrow from the idea only two layers (levels) and call them slightly different to express our specific context.
[2] https://orientdb.com/
[3] https://www.arangodb.com/
[4] We refer interested readers to the extended version [11] for technical details.
[5] But they can be restricted to intra-model only, like they are in existing MMDs.
[6] We assume re-use of existing single-model change propagation approaches and therefore focus on the novel problem of multi-model change propagation.

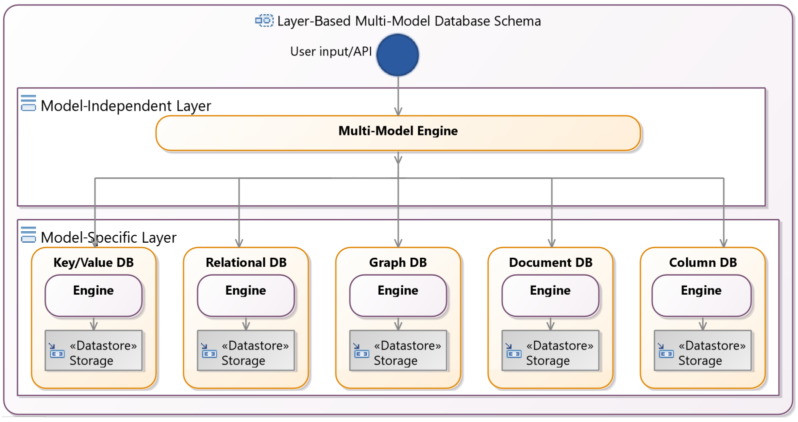

Figure 2: Architecture of a layer-based MMD

```
mmevolutionop ::= add | delete | rename | move | copy;

add    ::= "add" property "=" value [selection];
delete ::= "delete" property [selection];
rename ::= "rename" property "to" pname [selection];
move   ::= "move" property "to" kind [complexcond];
copy   ::= "copy" property "to" kind [complexcond];

selection   ::= "where" conds;
complexcond ::= "where" (joincond | conds |
                     (joincond "and" conds));

joincond ::= property "=" property;
conds    ::= cond {"and" cond};
cond     ::= property "=" value;

property ::= mname "." kname ["." pname];
kind     ::= mname "." kname;
mname    ::= identifier;
kname    ::= identifier;
pname    ::= identifier;
```

Figure 3: EBNF syntax of MMSEL

model where the entity occurs. We use this approach to detect the affected model in all modified functions. In Rule 8 we extend function $delete(\delta, \kappa)$ by key of the model δ which contains the entity with key κ. Rules 9, 10, and 11 add parameter δ to function get. All modified functions get load entity/entities from the specified model by key δ to the application space. Rule 12 is also extended by the model key δ to load the property from the specified model.

2.4 Reference Evolution in MMDs

We next discuss how referential integrity is maintained as schema modification operations are carried out.[7] In the first version of our solution, we consider the reference simply as a pointer from a property of a referencing entity to a property of a referenced entity. We describe the source or target of a reference by a triple *(model, entity, property)*. A reference is then represented as a pair $(source, target)$ and we assume that at least one model is able to store the pairs in a *reference space*.

Next we can divide operations into two groups: *Safe* operations (i.e., *add* and *copy*) do not trigger any reference updates, whereas *unsafe* operations (i.e., *delete*, *rename*, and *move*) can.

To avoid complex extensions and instead stay within the MMDPL framework, we internally represent a reference as a special type of entity. It has three properties: (1) the referenced property of the entity, (2) the key of the property in the MMD, and (3) an array of triples describing entities referencing it. Each triple in the array consists of three properties: a model, a kind, and a property.

During our analysis of reference migration, we discovered that WHERE conditions make the solution much more difficult. It is caused by the nature of MMDs which allow a user to move a subset of the properties. This behavior can split, delete or move completely an existing reference based on the affected set of the values. We introduce a solution for operations without WHERE conditions and keep it as an open challenge.

Another point to discuss is the behavior when a referenced property is removed. We have two options what can happen with the referencing property: (1) set to a default value, (2) delete the property. We decided to use the second approach, because it is a clear solution for the used models. (The first approach has to define what should be the behavior when a MMD contains an entity without a referencing property, as well as default values for all models. Also, the default value can be considered as a value of the property in an application so it can be confusing.)

The next step defines operations for creating and managing references in the MMDPL. We need to create a reference, store it, remove it and find it. Let *reference store model* (RSM) be a store which is able to persist the reference entities. We use the RSM to extend the MMDPL and define functions which help us to implement reference management in the MMSEL. Figure 5 shows the extension of the MMDPL which provides functions for the mentioned operations. We introduced a special type of entities for the references which is stored in the RSM, but we work with the type in the same way as with other database entities.

3 DEMONSTRATION OUTLINE

For the purpose of experimental evaluation of the above described ideas, we have implemented a first prototype called *MM-evolver*. The application is based on the .NET framework and is written in C#. To be able to experiment in the future with various models, not just those provided by a particular MMD, we created an *abstract* layer-based model, where each particular model can be represented by a separate DBMS. To test also this feature, in the first version we use MongoDB[8] and MariaDB[9] representing the document model and the relational model.

In our demo of *MM-evolver*, we build two use cases – one around real-world data, which is based on the Internet Movie Database (IMDb)[10], and the other on the multi-model benchmark UniBench[11]. As shown in Figure 6 for the first case, the data is stored both in the document model (such as `movies.Contributor` to the left), as well as in the relational model (`name_basics` to the right). To the top, we show an inter-model schema modification operation that we are about to execute. We further highlight the affected data (relational in red, document in blue). In interaction with our audience, we will gradually evolve the database state, simulating realistic demands. We intend to make the benefits of the declarative language evident, so that attendees get a clear picture how they would use it in practice. For each supported operation (both intra-model and inter-model), we demo:

(1) the state of the database before and after the change,

[7] Note that the earlier language NoSQLSEL does not consider references, because most NoSQL databases do not support them. Maintaining referential integrity is therefore another new contribution of *MM-evolver*.

[8] https://www.mongodb.com/
[9] https://mariadb.org/
[10] https://www.imdb.com/
[11] http://udbms.cs.helsinki.fi/?projects/ubench

Let dms be a DMS, ds be a database state, as be an application state, δ be a model key, and Ω be a data model. Let κ, κ' be entity keys. Let n, n' be property names, and let v be a property value. Symbol $\perp$ denotes an undefined value. Let π, π' be properties, i.e., mappings from property names to property values. *kind* : *Keys* $\mapsto$ *Kind* is a function that extracts the entity kind from a key. *model* : *Keys* $\mapsto$ *Data model* is a function that extracts the entity model from a key. Θ is a conjunctive query, and c is a string constant.

$$[\![empty()]\!](dms, ds, as) = (dms, ds, as) \tag{1}$$

$$[\![new(\kappa)]\!](dms, ds, as) = (dms, ds, as[\kappa \mapsto \emptyset]) \tag{2}$$

$$[\![new(\kappa, \pi)]\!](dms, ds, as) = (dms, ds, as[\kappa \mapsto \pi]) \tag{3}$$

$$[\![setProperty(\kappa, n, v)]\!](dms, ds, as \cup \{\kappa \mapsto \pi\}) = (dms, ds, as \cup \{\kappa \mapsto (\pi[n \mapsto v])\}) \tag{4}$$

$$[\![setProperty(\kappa, n, \kappa')]\!](dms, ds, \quad as \cup \{\kappa \mapsto \pi\} \cup \ \{\kappa' \mapsto \pi'\}) = (dms, ds, as \cup \{\kappa \mapsto (\pi[n \mapsto \pi'])\} \cup \{\kappa' \mapsto \pi'\}) \tag{5}$$

$$[\![removeProperty(\kappa, n)]\!](dms, ds, as \cup \{\kappa \mapsto \pi\}) = (dms, ds, as \cup \{\kappa \mapsto (\pi[n \mapsto \perp])\}) \tag{6}$$

$$[\![put(\delta, \kappa)]\!](dms \cup \{\delta \mapsto \Omega\}, ds, as \cup \{\kappa \mapsto \pi\}) = (dms \cup \{\delta \mapsto \Omega\}, ds[\{\kappa \mapsto \pi \mid model(\kappa) = \delta\}], as \cup \{\kappa \mapsto \pi\}) \tag{7}$$

$$[\![delete(\delta, \kappa)]\!](dms \cup \{\delta \mapsto \Omega\}, ds, as) = (dms \cup \{\delta \mapsto \Omega\}, ds[\{\kappa \mapsto \perp \mid model(\kappa) = \delta\}], as) \tag{8}$$

$$[\![get(\delta, \kappa)]\!](dms \cup \{\delta \mapsto \Omega\}, ds, as) = (dms \cup \{\delta \mapsto \Omega\}, ds, as \cup [\{\kappa \mapsto \pi \mid \kappa \mapsto \pi \in ds \wedge model(\kappa) = \delta\}]) \tag{9}$$

$$[\![get(\delta, kind = c)]\!](dms, ds, as) = (dms \cup \{\delta \mapsto \Omega\}, ds, as[\{\kappa \mapsto \pi \mid \kappa \mapsto \pi \in ds \wedge kind(\kappa) = c \wedge model(\kappa) = \delta\}]) \tag{10}$$

$$[\![get(\delta, kind = c \wedge \Theta)]\!](dms, ds, as) = (dms \cup \{\delta \mapsto \Omega\}, ds, as[\{\kappa \mapsto \pi \mid \kappa \mapsto \pi \in ds \wedge kind(\kappa) = c \wedge model(\kappa) = \delta \wedge [\![\Theta]\!](\kappa \mapsto \pi)\}]) \tag{11}$$

$$[\![getProperty(\delta, \kappa, n)]\!](dms \cup \{\delta \mapsto \Omega\}, ds, as \cup \{\kappa \mapsto (\{n \mapsto v\} \cup \pi) \mid \kappa \mapsto (\{n \mapsto v\} \cup \pi) \in ds \wedge model(\kappa) = \delta\}) = v \tag{12}$$

Figure 4: The commands for interfacing with the multi-model database

Let rs be a RSM, ρ_1,ρ_2 be keys in rs and η_1 its set of properties. Let dms be a DMS, ds be a database state, as be an application state. Let κ_1, κ_2 be entity keys. Let n_1 be property name. Let δ be a model key, c be a kind and v be an array of triples of *m*, *k*, and *p*. Symbol $\perp$ denotes an undefined value. Let π_1 be properties, i.e., mappings from property names to property values. key : *RSM keys* $\mapsto$ *model keys* is a function that extracts the entity key from a reference store model key.

$$[\![newReference(\kappa_1)]\!](dms, ds, as, rs) = (dms, ds, as[\{\rho_1 \mapsto \perp \mid key(\rho_1) = \kappa_1\}], rs) \tag{13}$$

$$[\![putReference(\rho_1)]\!](dms, ds, as \cup \{\rho_1 \mapsto \eta_1\}, rs) = (dms, ds, as \cup \{\rho_1 \mapsto \eta_1\}, rs[\rho_1 \mapsto \eta_1]) \tag{14}$$

$$[\![getReference(\kappa_1)]\!](dms, ds, as, rs) = (dms, ds, as[\{\rho_1 \mapsto \eta_1 \mid \rho_1 \mapsto \eta_1 \in rs \wedge key(\rho_1) = \kappa_1\}], rs) \tag{15}$$

$$[\![getReferencedBy(\delta, c_1, n_1)]\!](dms, ds, as, rs) = (dms, ds, as[\{\rho_1 \mapsto \eta_1 \mid \rho_1 \mapsto \eta_1 \in rs \wedge \{''s'', v\} \in \eta_1 \wedge \{''m'' : \delta,'' k'' : c,'' p'' : n_1\} \in v\}], rs) \tag{16}$$

$$[\![deleteReference(\kappa_1)]\!](dms, ds, as, rs) = (dms, ds, as, rs[\{\rho_1 \mapsto \perp \mid key(\rho_1) = \kappa_1\}]) \tag{17}$$

$$[\![renameReference(\kappa_1, \kappa_2)]\!](dms, ds, as, \quad rs \cup \{\rho_1 \mapsto \eta_1 \mid \rho_1 \mapsto \eta_1 \wedge key(\rho_1) = \kappa_1\}) = (dms, ds, as, rs[\{\rho_2 \mapsto \eta_1 \mid key(\rho_2) = \kappa_2\}]) \tag{18}$$

Figure 5: Dedicated commands for manipulating references in the multi-model database

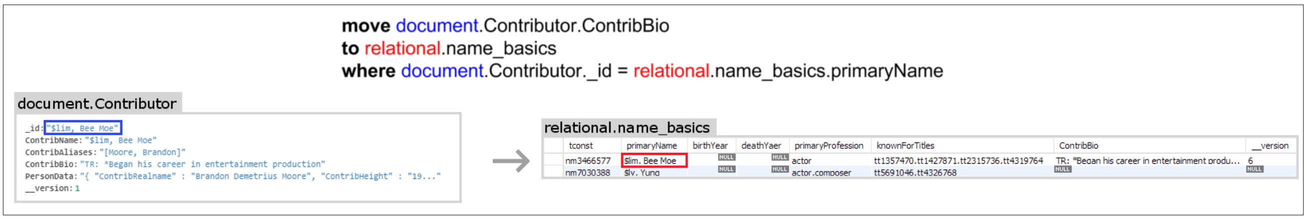

Figure 6: Carrying out an inter-model schema modification operation in *MM-evolver*

(2) the number of affected entities, i.e., those changed during the execution of an operation,
(3) the number of targeted entities, i.e., those that correspond to the change request, and
(4) the generated code which propagates these changes.

Interested attendees can experiment with *MM-evolver*, issuing their own operations and thus evaluating our approach.

Acknowledgements and Remarks

This paper is based on Michal Vavrek's Master thesis [11]. In part, this project was funded by the MŠMT ČR project SVV 260451 and by the *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation), grant #385808805.

REFERENCES

[1] Bob Bryla and Kevin Loney. 2013. *Oracle Database 12C The Complete Reference* (1st ed.). McGraw-Hill Osborne Media. 1472 pages.
[2] Couchbase 2018. https://www.couchbase.com/ [Online; Accessed 2-February-2018].
[3] Donald Feinberg, Merv Adrian, Nick Heudecker, Adam M. Ronthal, and Terilyn Palanca. [n. d.]. *Magic Quadrant for Operational Database Management Systems.* Gartner. 12 October 2015.
[4] Zhen Hua Liu and Dieter Gawlick. 2015. Management of Flexible Schema Data in RDBMSs – Opportunities and Limitations for NoSQL. In *CIDR '15*. Online Proceedings, Asilomar, California, USA.
[5] Jiaheng Lu and Irena Holubová. 2017. Multi-model Data Management: What's New and What's Next?. In *EDBT '17*. OpenProceedings, Venice, Italy, 602–605.
[6] Jiaheng Lu, Irena Holubová, and Bogdan Cautis. 2018. Multi-model Databases and Tightly Integrated Polystores: Current Practices, Comparisons, and Open Challenges. In *CIKM '18*. ACM, Torino, Italy, 2301–2302.
[7] Martin Nečaský, Jakub Klímek, Jakub Malý, and Irena Mlýnková. 2012. Evolution and Change Management of XML-based Systems. *J. Syst. Softw.* 85, 3 (March 2012), 683–707.
[8] Stefanie Scherzinger, Thomas Cerqueus, and Eduardo Cunha de Almeida. 2015. ControVol: A framework for controlled schema evolution in NoSQL application development. In *ICDE '15*. IEEE Computer Society, Seoul, South Korea, April 13-17, 1464–1467.
[9] Stefanie Scherzinger, Meike Klettke, and Uta Störl. 2013. Managing Schema Evolution in NoSQL Data Stores. In *DBPL'13*. arXiv, Riva del Garda, Trento, Italy.
[10] Michael Stonebraker and Ugur Cetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE '05*. IEEE Computer Society, Washington, DC, USA, 2–11.
[11] M. Vavrek. 2018. *Evolution Management in NoSQL Document Databases*. Master Thesis. Charles University in Prague, Czech Republic. http://www.ksi.mff.cuni.cz/~holubova/dp/Vavrek.pdf.