Koupil and Holubová *Journal of Big Data* (2022) 9:61
https://doi.org/10.1186/s40537-022-00613-3

Journal of Big Data

RESEARCH Open Access

A unified representation and transformation of multi-model data using category theory

Pavel Koupil* and Irena Holubová

*Correspondence:
pavel.koupil@matfyz.cuni.cz
Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Abstract

The support for multi-model data has become a standard for most of the existing DBMSs. However, the step from a conceptual (e.g., ER or UML) schema to a logical multi-model schema of a particular DBMS is not straightforward. In this paper, we extend our previous proposal of multi-model data representation using category theory for transformations between models. We introduce a mapping between multi-model data and the categorical representation and algorithms for mutual transformations between them. We also show how the algorithms can be implemented using the idea of wrappers with the interface published but specific internal details concealed. Finally, we discuss the applicability of the approach to various data management tasks, such as conceptual querying.

Keywords: Multi-model data, Category theory, Model transformations

Introduction

The *variety* feature of Big Data inciting the so-called *multi-model data* has opened a challenging direction of data management.

Example 1.1 An example of a multi-model scenario is provided in Fig. 1. The relational model (violet) contains general information about customers, whereas the graph model (blue) captures their mutual friendship. The document model (green) maintains orders bounded with particular customers using the wide-column model (red). The key/value model (yellow) bears information about customers' shopping carts. A cross-model query over such data might, e.g., be "For each customer who lives in Prague, find a friend who ordered the most expensive product among all customer's friends." [1] □

In general, there are two approaches to ensure the storage and processing of multi-model data in their most native and thus most efficient environment. The (primarily) academia-driven approaches, currently represented mainly by *polystores* [2], are based on the idea of *polyglot persistence*, i.e., the usage of a *mediator* to manage a set of underlying database management systems (DBMSs), each being the best suitable candidate for a particular data model. On the other hand, there are (industry-driven) *multi-model DBMSs* [3] that offer the support of multiple models under the hood of

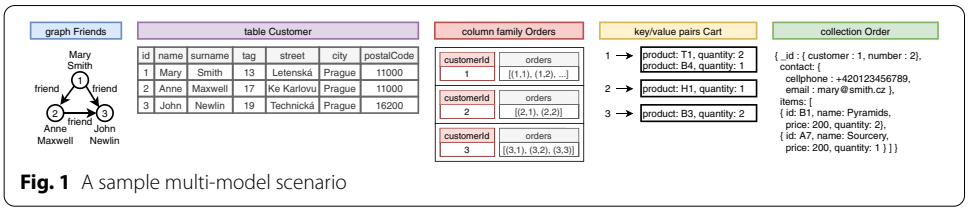


Fig. 1 A sample multi-model scenario

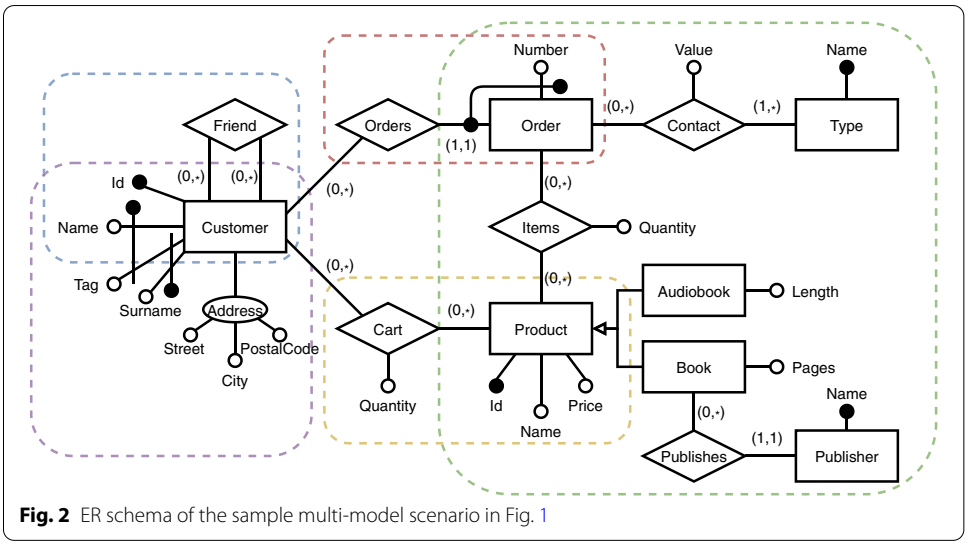


Fig. 2 ER schema of the sample multi-model scenario in Fig. 1

a single system, treating all the data models as first-class citizens [4]. Currently, more than 20 representatives of multi-model DBMSs exist, involving well-known traditional, relational and novel NoSQL systems. In contrast, more vendors decide to follow the Gartner predictions [5] of supporting multiple data models.

On the other hand, such a situation is difficult for users who want to develop a multi-model database application. The standard recommendations would be to first create a conceptual schema (e.g., using ER or UML modelling languages).

Example 1.2 In Fig. 2, we depict an ER schema of the multi-model scenario from Fig. 1. □

There are verified means of transforming such a schema into, e.g., the relational model schema. (More-or-less) according to its well-defined standard, the existing relational DBMSs support this model. However, the step from an ER/UML conceptual model to virtually any possible (yet not standardized) combination of multiple logical models is not straightforward, mainly because the combined models (or respective systems) often have contradictory features. For example, there are structured, semi-structured, and unstructured formats; there are systems based on strong or eventual consistency; there are schema-less, schema-full, and schema-mixed storage strategies, etc.

For this purpose, we need a unifying representation that would allow us to:

1. Capture all the existing models, preferably in the same and definitely in a standard way;
2. Query across multiple interconnected models efficiently;
3. Perform correct and complete evolution management, i.e., propagation of changes;
4. Enable data migration without complex reorganizations; and
5. Permit integration of new data models.

Although both ER and UML (class diagrams in particular) are strong enough to cover some of these points, their primary purpose is different and not that wide. As stated in [6], we need "*a theory that is the basis upon which a designer can build a consistent schema that can be understood by other designers and consistently rebuilt during redesign or schema development*". Hence, in paper [7], we have proposed a solution based on *category theory* [8], "*the most general and abstract branch of pure mathematics*" [9] which has successfully been applied in computer science and namely data management, too. It is a theory sufficiently general for the multi-model situation. It provides a strong mathematical background for further data management, such as transformations between the models, cross-model querying, multi-model evolution management, etc. We have proposed a *schema category* and an *instance category* for the representation of multi-model data structures and their particular instances, as well as an algorithm for the transformation of an ER schema to a schema category.

In this paper, we further extend the idea and show how the currently popular data models (and their combinations) can be represented using category theory. The main contributions of the paper can be summed up as follows:

- We provide a more general definition of both schema and instance categories, which enable a unified and sufficiently general representation of schemas and instances of multi-model data.
- We introduce mapping between the input data and the categorical representation using the notion of an *access path* that bears information about the categorical representation of any object.
- We introduce transformation algorithms that transform the input data to the categorical representation and vice versa. The algorithms are sufficiently generic to cover all currently popular models and their combinations.
- We show how the proposed algorithms can be comfortably implemented using wrappers that hide the specifics of particular DBMSs.
- We discuss the applicability of the proposed approach in further data management tasks, such as querying or data migration/evolution.

The rest of the paper is structured as follows: In "Unified view of multi-model data" section, we provide a unified view of multi-model data which enables the further general description of the proposed ideas. We also recall the basic terms from the category theory used in the rest of the text and we describe our proposal of the schema and instance categories, including their novel extensions. "Category-to-data mapping" section introduces the mapping of constructs of particular models to their categorical representation using access paths, i.e., a novel concept that enables the capture of

the necessary information for all considered models and their specifics universally. Next, in "Transformations" section , we focus on the algorithms for the transformation of multi-model data to the categorical representation and vice versa. We provide pseudocodes of the algorithms and an explanatory description with examples. In "Framework MM-cat" section, we describe the specifics of the implementation of the proposed algorithms—a framework called *MM-cat*. We describe its architecture and implementation decisions and the performance of the implemented algorithms, including some technical tricks. In "Benefits of category theory" section we discuss the general benefits of the application of category theory for multi-model data representation and we provide an example in the case of multi-model querying. In "Related work" section, we overview the related work and its drawbacks reflected in our approach. We conclude and outline future work in "Conclusion" section.

Unified view of multi-model data

First of all, we need to be able to "grasp" the specifics of various data models in a unified way. In this section, we first unify the terminology. Next, we introduce the basic concepts of category theory used in the rest of the proposal. We also introduce the idea of an extended categorical representation of multi-model data.

In the rest of our work, we consider the following popular data models: relational, key/value, document, wide column, and graph, i.e., we support all currently popular structured and semistructured data to cover all combinations of models used in the existing popular multi-model systems[1] Unstructured data can be treated in the same way as key/value data, where the value part is considered as a black box.

Since the terminology within the considered models differs, first we provide a unification used throughout the text in Table 1. (As we can see, we also incorporated the array and RDF models since the proposed approach applies to them, too.)

The terminology is apparent in most cases, but some specific situations need commentary:

- Probably the most protuberant is the graph model, whose features are the most specific. We assume that a kind is represented by a unique label that determines a set of related nodes or edges. A record is either a node or an edge.
- The document and column model can involve a hierarchical structure. Therefore, the properties (fields) can appear at various levels. In the case of the document model, it can be on any level. In the case of the column model, there can be a second level grouping the selected columns to a super column.[2] In the other models, the structures are always single-level.
- We distinguish between homogeneous and heterogeneous arrays. In the former case, an array should contain fields of the same type. In the latter case, which is allowed only in the document model, an array can contain fields of multiple types. Only in the case of the document model, the type of an array item can be complex (i.e., rep-

[1] https://db-engines.com/en/ranking.

[2] Note that in some systems, e.g., *Cassandra*, even multiple levels of nesting are allowed. However, we can consider this case as a multi-model extension.

resent nested documents); in all other cases, only arrays of simple (scalar) types are allowed.

Despite this unification, we still have to bear in mind important differences between the models. One of the core classifications assumes the following cases:

- *Aggregate-oriented models* (key/value, document, column): These models primarily support the data structure of an *aggregate*, i.e., a collection of closely related (semi-) structured objects we want to treat as a unit. In the traditional relational world, we would speak about de-normalization.
- *Aggregate-ignorant models* (graph, relational, RDF, array): These models are not primarily oriented to the support of aggregates. The relational world strongly emphasizes the normalization of structured data, whereas the graph model is in principle a set of flat objects mutually linked by any number of edges.

We will show later on that these different perspectives will have an impact on the way how the algorithms we introduce will operate.

Basic concepts of category theory

Category theory is a branch of mathematics that attempts to formalize various (not only) mathematical structures and their mutual relationships. Formally, a category $\mathbf{C} = (\mathcal{O}, \mathcal{M}, \circ)$ consists of a set of objects $\mathcal{O}$, also alternatively denoted as $Obj(\mathbf{C})$, a set of morphisms $\mathcal{M}$, alternatively $Hom(\mathbf{C})$[3], and a composition operation $\circ$ over the morphisms.[4] A category as a whole can be visualized in the form of a multigraph, where category objects act as vertices and category morphisms as directed edges.

Each morphism is modeled and depicted as an *arrow* $f : A \to B$, where $A, B \in Obj(\mathbf{C})$, and A is referenced to as a *domain* and B as a *codomain*, both denoted as *f.dom* and *f.cod*, respectively. Whenever $f, g \in Hom(\mathbf{C})$ are two morphisms $f : A \to B$ and $g : B \to C$, it must hold that $g \circ f \in Hom(\mathbf{C})$, i.e., morphisms can be composed using the $\circ$ operation and the composite $g \circ f$ must also be a morphism of the category. Besides this *transitivity* property, $\circ$ must also be *associative*, i.e., $h \circ (g \circ f) = (h \circ g) \circ f$ for any suitable morphisms $f, g, h \in Hom(\mathbf{C})$ such that $f : A \to B, g : B \to C$, and $h : C \to D$. Finally, for every object A, there must exist an *identity* morphism 1_A such that $f \circ 1_A = f = 1_B \circ f$ for any $f : A \to B$, and so acts as a unit element with respect to the composition operation.

Example 2.1 In Fig. 4, we can see a graphical representation of a simple category having three objects *a*, *b*, *c*, two morphisms *f*, *g*, their composition $g \circ f$, and identity morphisms id_a, id_b, id_c. □

[3] The common notation *Hom* results from the fact that morphisms are often called *homomorphisms*.

[4] A category, where $Obj(\mathbf{C})$ and $Hom(\mathbf{C})$ are sets, is denoted as *small*. There are also *large* categories, but we will not need them in our approach.

Example 2.2 **Set** (as widely denoted) is a category where objects are arbitrary sets (not necessarily finite), and morphisms are functions between them (not necessarily injective nor surjective), together with the traditionally understood composition of functions and identities.

Similarly, **Rel** is a category where objects represent sets, and morphisms are binary relations over these sets. As for the composition $g \circ f$ for morphisms $f : A \to B$ and $g : B \to C$, it holds that $(a, c) \in g \circ f$ for any $a \in A$ and $c \in C$ whenever there exists at least one value $b \in B$ such that $(a, b) \in f$ and $(b, c) \in g$. □

Even though objects and morphisms in real-world categories tend to be sets of certain items and functions between them, both objects and morphisms may represent abstract entities of any kind and internal content. Not just in the context of our approach, it is worth focusing on categories derived from graphs, as well as categories built on top of other categories.

Example 2.3 Having a graph $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of directed edges, we could define another category where objects are the original vertices and morphisms simply the original edges. Composition $\circ$ produces a kind of collapsed shortcut for concatenated directed paths consisting of individual edges, identity morphisms work as loops.

However, such a structure may not always define a category, since it may happen that for any two edges (morphisms) $f = (a, b)$ and $g = (b, c) \in E$ the composite $g \circ f = (a, c) \notin E$, i.e., the composed edge may not be in the graph. As a consequence, not every graph necessarily forms a category. □

Categories themselves can also be mutually mapped via structure-preserving mappings called *functors*. A functor F is a mapping between categories $\mathbf{C}_1 = (\mathcal{O}_1, \mathcal{M}_1, \circ_1)$ and $\mathbf{C}_2 = (\mathcal{O}_2, \mathcal{M}_2, \circ_2)$ associating each object $A \in Obj(\mathbf{C}_1)$ with an object $F(A) \in Obj(\mathbf{C}_2)$, and each morphism $f : A \to B \in Hom(\mathbf{C}_1)$ with a morphism $F(f) : F(A) \to F(B) \in Hom(\mathbf{C}_2)$. We must also ensure that identity morphisms and compositions are both preserved. In particular, $F(1_A) = 1_{F(A)}$ for each $A \in Obj(\mathbf{C}_1)$, and $F(g \circ_1 f) = F(g) \circ_2 F(f)$ for any $f, g \in Hom(\mathbf{C}_1)$, $f : A \to B$ and $g : B \to C$, respectively.

Categorical representation of multi-model data

The idea to define a unified structure for the representation of multi-model data based on category theory was already introduced in paper [7]. In particular, notions of a *schema category* describing the conceptual structure (schema) of the data and an *instance category* encompassing a particular data instance conforming to a given schema category were described. We also introduced an algorithm for transforming an ER schema to a corresponding schema category so that users can easily understand the categorical approach in terms of a well-known conceptual modeling strategy. Nevertheless, schema categories can also be designed directly from scratch without creating ER schemas first.

This section provides an extended version of the definitions of both the schema and instance categories. The core idea remains the same, but several changes were introduced to increase their expressive power.

Schema category

Schema category $\mathbf{S}$ is defined as a tuple $(\mathcal{O}_\mathbf{S}, \mathcal{M}_\mathbf{S}, \circ_\mathbf{S})$. Borrowing the ER terminology, objects in $\mathcal{O}_\mathbf{S}$ correspond to individual entity types, attributes, and relationship types. Hence, if $\mathbf{S}$ is derived from an ER schema (but it does not have to be), we can distinguish *entity*, *attribute*, and *relationship* objects, and, analogously, *attribute*, *relationship*, and *hierarchy* morphisms. This distinction is introduced solely to increase comprehensibility since objects and morphisms of all kinds are always treated and processed the same way. Morphisms in $\mathcal{M}_\mathbf{S}$ connect appropriate pairs of objects. The explicitly defined morphisms are denoted as *base*, those obtained via the composition $\circ$ as *composite*.

Each object $o \in \mathcal{O}_\mathbf{S}$ is internally modeled as a tuple (*key*, *label*, *superid*, *ids*), where $key \in \mathbb{O}$ is an automatically assigned internal identity ($\mathbb{O} \subseteq \mathbb{N}$ being their domain[5]), *label* is an optional user-defined name (e.g., name of the corresponding entity type) or $\perp$ when missing, $superid \neq \emptyset$ is a set of attributes (each corresponding to a signature of a base or composite morphism as they are introduced later on)[6] forming the actual data contents a given object is expected to have, and $ids \subseteq \mathcal{P}(superid)$, $ids \neq \emptyset$ is a set of particular identifiers (each modeled as a set of attributes) allowing us to uniquely distinguish such individual data instances. It holds that $superid \supseteq \bigcup_{id \in ids} id$. In the case of entity or attribute objects, equality holds.

Each morphism $m \in \mathcal{M}_\mathbf{S}$ is a tuple (*signature*, *dom*, *cod*, *min*, *max*). $signature \in \mathbb{M}^*$ allows us to mutually distinguish all morphisms except the identity ones. $\mathbb{M}^*$ is a set of all the possible strings over the alphabet $\mathbb{M}$, i.e., all possible sequences of symbols from $\mathbb{M}$ connected using the $\cdot$ operation (e.g., 15, 3.7.5, or ε being a metasymbol representing an empty string). $signature \in \mathbb{M}$ is used for the base morphisms. $signature \in \mathbb{M}^* \setminus (\mathbb{M} \cup \{\varepsilon\})$ is used for the composite morphisms allowing their decomposition to base morphisms, which is directly related to the definition of the $\circ$ operation itself. $signature = \varepsilon$ is used for identity. *dom* and *cod* represent the domain and codomain of the morphism, whereas the triple (*signature*, *dom*, *cod*) enables one to distinguish also the identity morphisms. Finally, $min \in \{0, 1\}$ and $max \in \{1, *\}$ allow us to express constraints on minimal/maximal numbers of occurrences, analogously as we can do in the traditional ER modeling[7].

Identity morphism for an object $o \in \mathcal{O}_\mathbf{S}$ is defined as $1_o = (\varepsilon, o, o, 1, 1)$. Whenever $m_1 = (signature_1, dom_1, cod_1, min_1, max_1)$ and $m_2 = (signature_2, dom_2, cod_2, min_2, max_2)$ are two morphisms $m_1, m_2 \in \mathcal{M}_\mathbf{S}$, their composite is evaluated as $m_2 \circ_\mathbf{S} m_1 = (signature, dom_1, cod_2, min, max)$, where $signature = signature_2 \cdot signature_1$ except the case when a non-identity morphism is composed with an identity one (in any

[5] We assume that these keys are assigned automatically, e.g., by a tool supporting the process of creation of schema categories or their transformation from ER schemas. Though we have chosen natural numbers, this particular decision has no impact on the definitions.

[6] Not necessarily corresponding to attributes from ER, though in some cases they may coalesce and mutually correspond to each other.

[7] For the sake of easier explanation, we only use these basic types of cardinalities. The proposed algorithms can be extended to other commonly used ones too.

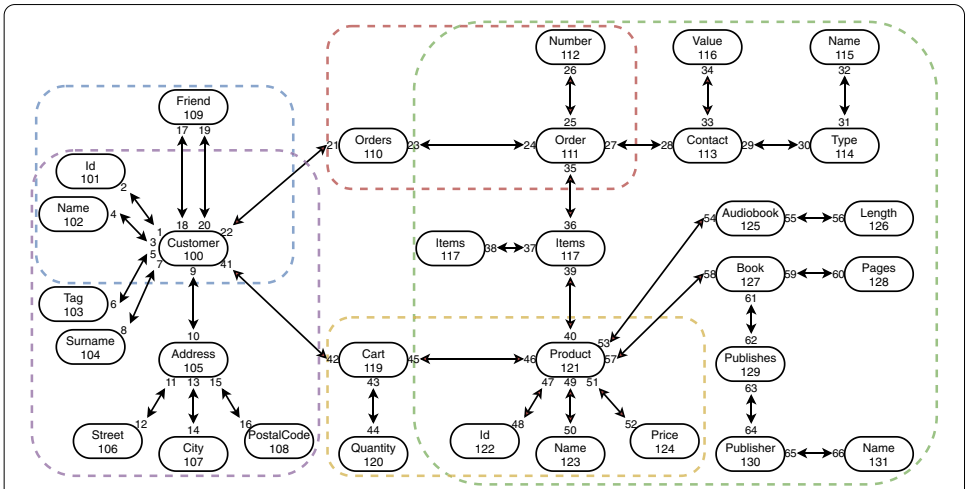


Fig. 3 Schema category extracted from the sample ER schema in Fig. 2

order). As for the cardinalities, $min = \min(min_1, min_2)$ and $max = \max(max_1, max_2)$, i.e., the lowest of limits for the lower bound and the highest for the upper one are chosen.

Finally, for technical reasons, whenever $m : X \to Y$ is a non-identity morphism between two particular objects, there must also exist its *dual* morphism $m^{-1} : Y \to X$. Its purpose is to restrain the opposite direction of the same relationship between a given pair of objects since morphisms are always directed. Therefore, both directions need to be treated separately.

The algorithm that transforms an input ER schema to schema category [7] creates an object for each entity type, relationship type, and attribute (one for an attribute as a whole, additional objects for its subattributes in the case of structured attributes). Labels, identifiers, and cardinalities are taken over from the respective ER constructs. ISA hierarchies and weak entity types are processed in the correct order, i.e., starting from the root/strong entity types and following the rules for inheriting identifiers. As we have mentioned, the schema category can be created directly, and thus there may exist morphisms between any kind of objects, depending on the respective data model it represents. If a schema category $\mathbf{S}$ is derived from an ER schema, the morphisms correspond to its structure. Hence, there are morphisms, e.g., between entity and attribute objects, but not between two attribute objects.

Example 2.4 The schema category of ER model in Fig. 2 is depicted in Fig. 3. Each object is represented as a node labeled with *key* and *label*. Morphisms are represented as directed edges labeled with *signature* at its beginning. To simplify the figure, we do not depict the identity and composite morphisms and the cardinalities of the morphisms (which correspond to those in the ER schema). We also do not depict *superid* and *ids* of objects. And, for the sake of clarity of further examples, the keys of objects are ≥ 100, whereas signatures of base morphisms are < 100.[8]

[8] In general, the identifiers can be assigned randomly. To speed up the access, we assigned each two mutually dual morphisms with respective positive and negative integers in the implementation.

Let us look closely at the structure of the selected objects. For example, object *Product* has a simple identifier *id*. Thus its full categorical representation is:

$\{121, \text{“}Product''\text{”}, \{47\}, \{\{47\}\}\}$

Considering object *Order* with a mixed weak identifier, its categorical representation is:

$\{111, \text{“}Order''\text{”}, \{25, 1.21.24\}, \{\{25, 1.21.24\}\}\}$

Object *Customer* having simple identifier *id*, and two composed and even overlapping identifiers *(name, tag)* and *(surname, tag)*. Therefore, its categorical representation is:

$\{100, \text{“}Customer''\text{”}, \{1, 3, 5, 7\}, \{\{1\}, \{3, 5\}, \{7, 5\}\}\}$

Relationship object *Publishes* has the following categorical representation:

$\{129, \text{“}Publishes''\text{”}, \{47.58.62, 65.53\}, \{\{65.53\}\}\}$

Signature 47.58.62 leads to object *Id* identifying object *Book* (see the ISA hierarchy in the ER model), while signature 65.53 points to object *Name* identifying object *Publisher*. Due to cardinalities in relationship *Book-Publishes-Publisher*, the minimal identifier required to identify the relationship *Publishes* is {65.53}, meaning that a single publisher may publish many books. However, a particular book is published only by a single publisher. □

Instance category

While the purpose of the schema category $\mathbf{S}$ is to describe the structure of the data at the conceptual layer, *instance category* $\mathbf{I}$ is a data structure capable of holding the actual data stored within a (set of) DBMS(s). Each instance category $\mathbf{I} = (\mathcal{O}_\mathbf{I}, \mathcal{M}_\mathbf{I}, \circ_\mathbf{I})$ represents a particular data instance conforming to a particular schema category $\mathbf{S}$. It permits us to encompass all data valid against $\mathbf{S}$ stored in the database at a selected moment. When the data is modified (within the restrictions given by $\mathbf{S}$), a new instance category is obtained.

Objects $\mathcal{O}_\mathbf{I}$ as well as morphisms $\mathcal{M}_\mathbf{I}$ directly correspond to the objects $\mathcal{O}_\mathbf{S}$ and morphisms $\mathcal{M}_\mathbf{S}$ in schema category $\mathbf{S}$, respectively. Hence, both categories intentionally have the same structure, they only differ in what their objects and morphisms represent. Assuming that $\mathbb{V}$ is the domain of all possible values of attributes, object $o_\mathbf{I} = \{t_1, t_2, \ldots, t_n\} \in \mathcal{O}_\mathbf{I}$ for some $n \in \mathbb{N}$ is modeled as a set of tuples, each represented as a function $t_i : superid \rightarrow \mathbb{V}$ for any $i \in \mathbb{N}, 0 < i \leq n$. The tuples are unordered, unique, and with all attributes specified. The particular set of tuples that are used for object $o_\mathbf{I} \in \mathcal{O}_\mathbf{I}$ is called *active domain* of $o_\mathbf{I}$.

Since *ids* is a set of identifiers defined in the corresponding schema category object $o_\mathbf{S}$, it must hold that each identifier $id \in ids$ has its identification ability, i.e., the cardinality of $o_\mathbf{I}$ must not change when unique tuples projected only to the attributes of a given identifier would be retrieved.

Example 2.5 An instance object $Customer_{\mathbf{I}}$ for $Customer_{\mathbf{S}}$ from Fig. 3 can, for example, be:

$Customer_{\mathbf{I}} = \{$

$\{(1, 1), (3, Mary), (5, 13), (7, Smith)\},$

$\{(1, 2), (3, Anne), (5, 17), (7, Maxwell)\},$

$\{(1, 3), (3, John), (5, 19), (7, Newlin)\}\}$

The tuples form the active domain of $Customer_{\mathbf{I}}$. □

Morphisms act as binary relations, i.e., they abide by the principles of the **Rel** category (see Example 1.4). In particular, having a morphism $m_{\mathbf{I}} \in \mathcal{M}_{\mathbf{I}}$, $m_{\mathbf{I}} : o_1 \rightarrow o_2$ for some objects $o_1, o_2 \in \mathcal{O}_{\mathbf{I}}$, it must then hold that $m_{\mathbf{I}} \subseteq o_1 \times o_2$. Moreover, the cardinality restrictions *min* and *max* imposed by the corresponding schema category morphism $m_{\mathbf{S}}$ must also be satisfied. It means that $\forall\, t_1 \in o_1$ it must hold that $|\{t_2 \mid t_2 \in o_2, (t_1, t_2) \in m_{\mathbf{I}})\}| = c$ must be within the cardinality boundaries. Identity morphism 1_o for each object $o \in \mathcal{O}_{\mathbf{I}}$ is defined as a function (i.e., a special case of a more generic relation) $1_o = \{(t, t) \mid t \in o\}$. The composition operation $\circ_{\mathbf{I}}$ corresponds to the composition in **Rel**.

Example 2.6 Consider object $Customer_{\mathbf{I}}$ from Example 1.7 and object $Surname_{\mathbf{I}}$ with the following active domain:

$Surname_{\mathbf{I}} = \{$

$\{(\epsilon, Smith)\},$

$\{(\epsilon, Maxwell)\},$

$\{(\epsilon, Newlin)\}\}$

Note that since $Customer_{\mathbf{I}}$ is attribute object, its $superid = \{\epsilon\}$, i.e., ϵ represents the identity morphism.

Morphism $7_{\mathbf{I}} : Customer_{\mathbf{I}} \rightarrow Surname_{\mathbf{I}}$ has the following set of relations[9]:

$7_{\mathbf{I}} = \{$

$(\{(1, 1), (3, Mary), (5, 13), (7, Smith)\}, \{(\epsilon, Smith)\}),$

$(\{(1, 2), (3, Anne), (5, 17), (7, Maxwell)\}, \{(\epsilon, Maxwell)\}),$

[9] Note that the morphisms are internally implemented as pairs of pointers to the respective objects representing data. So, there is no data duplication as might be indicated by the example.

Table 1 Unification of terms in popular models

Unifying term	Relational	Array	Graph	RDF	Key/value	Document	Column
Kind	Table	Matrix	Label	Set of triples	Bucket	Collection	Column family
Record	Tuple	Cell	Node/edge	Triple	Pair (key, value)	Document	Row
Property	Attribute	Attribute	Property	Predicate	Value	Field	Column
Domain	Data type	Data type	Data type	IRI/literal/ blank node	–	Data type	Data type
Value	Value	Value	Value	Object	Value	Value	Value
Identifier	Key	Coordinates	Identifier	Subject	Key	Identifier	Row key
Reference	Foreign key	–	–	–	–	Reference	–
Array	–	–	Array	–	Array	Array	Array
Structure	–	–	–	–	Set/ZSet/ Hash	Nested document	Super column

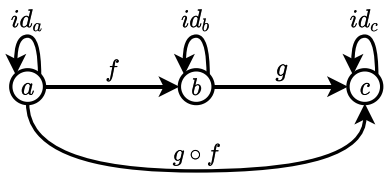


Fig. 4 An example of a category

$$(\{(1,3),(3,John),(5,19),(7,Newlin)\},\{(\epsilon,Newlin)\})\} \qquad \square$$

Having a schema category $\mathbf{S}$ and a particular instance category $\mathbf{I}$, we can introduce a pair of functors $Schm_{\mathbf{I}} : \mathbf{I} \rightarrow \mathbf{S}$ and $Inst_{\mathbf{I}} : \mathbf{S} \rightarrow \mathbf{I}$ using which we will be able to retrieve the corresponding counterparties.

Category-to-data mapping

Having defined a schema category, in this section we specify its mapping to the underlying (set of) DBMS(s), i.e., we describe how the actual data permitted by a given schema category are supposed to be stored within the data structures provided by the underlying database(s). Although this mapping can also be described directly, we assume that the whole decomposition process is aided by a tool that enables us to visualize and process schema categories, e.g., using a tool called *MM-cat* which we introduce in "Framework MM-cat" section.

This section aims to describe how the mappings are intended to be created and formalized. After an informal outline of the basic principles, we provide a formal definition of these mappings ("Formal Definitions" section), and we introduce an alternative way that these mappings can be visualized or even directly created by users using a textual representation ("JSON-like Representation' section).

The decomposition (which can be both partial and overlapping) is defined via a set of *mappings*, each describing where and how data instances of one schema category object or base morphism—possibly together with other data from neighboring or

even more remote objects or morphisms—are stored as individual records within a given kind (recall Table 1) in a particular underlying DBMS (i.e., as rows within a table in the traditional relational model, JSON documents within a collection in the document model, etc.).

During the decomposition process, the user is expected to create individual mappings (i.e., create individual kinds and define the internal structure of their records) iteratively, one by one. This means a particular DBMS needs to be selected first, so that a new kind can be introduced and all its characteristics specified. Besides the name of a given kind, one object or base morphism from the schema category is selected and appointed as the *root* object/morphism for a given kind, representing its initial context. Next, the user specifies the internal structure of the records, starting with the top-level properties and, optionally, continuing with their recursively nested properties.

The specification of a property (at any level) consists of its name and structure, which must follow the rules and limitations imposed by the particular model. For example, in the case of the relational model, the level of nesting cannot be greater than 1, properties cannot be multi-valued, as well as the names of the properties (columns) must be unique. Finally, at least one root object identifier must be covered by the involved properties. Similarly, suppose a given kind also has a root morphism. In that case, it must involve at least one identifier of both its domain and its codomain, i.e., both objects participating in the relationship given by the morphism.

When specifying a child property, there can occur three situations where the property can occur:

- A child property is a direct neighbor in the graph of the schema category, i.e., it is accessible via a base morphism.
- A child property is *inlined* from a more distant position, i.e., it is accessible via a composite morphism. Since more than one path may exist between two objects, the particular path, i.e., the composition of morphisms, must be denoted.
- A child property is defined as *auxiliary*, e.g., for grouping related properties. Hence, the respective object does not exist in the schema category.

When choosing a name of a property, there can also occur several situations:

- *Inherited*: The name of a schema category object is reused.
- *User-defined*: A completely new name is explicitly specified by the user.
- *Anonymous*: The name is entirely omitted in case no name is needed or permitted (e.g., for elements of an array in JSON).
- *Dynamically derived*: The name is derived from a particular instance of a schema category object.

Example 3.1 Consider the document collection *Order* in Fig. 1. For example, properties *customer* or *price* have user-defined names (different from the ones used in the ER schema and schema category in Figs. 2 and 3). Child property of property *items* has an anonymous name.

A dynamically derived name of a property can be seen in the case of child properties of property *contact*. Name and value of the contact are specified in respective attributes *Name* of entity type *Type* and *Value* of relationship type *Contact*. In schema category this can be done via a composite morphism which corresponds to the composition of respective morphisms on the path from node *Contact* to nodes *Name* and *Value*. □

Finally, the value of a property can be of the following two possible types:

- *Simple*, i.e., a single atomic value.
- *Complex* which encompasses a list or a set of child properties, i.e., an *array* or a *structure*.

Example 3.2 A sample decomposition is presented in Figs. 2 and 3 using the colors from Fig. 1. □

Formal definitions

More formally, the intended database decomposition is a set $\mathfrak{M}$ of mappings in a form of a tuple $(\mathcal{D}, name_\kappa, root_\kappa, morph_\kappa, pkey_\kappa, ref_\kappa, P_\kappa)$, each introducing one particular kind κ and describing the expected internal structure and contents of its records as follows:

- $\mathcal{D}$ denotes a particular DBMS, e.g., using a connection string.
- $name_\kappa$ is a name of kind κ.
- $root_\kappa \in \mathcal{O}_\mathbf{S}$ is a root object associated with κ.
- $morph_\kappa \in \mathcal{M}_\mathbf{S} \cup \{null\}$ is an optional root morphism associated with κ. It cannot be an identity morphism. If $morph_\kappa \neq null$, then $morph_\kappa.dom = root_\kappa$.
- $pkey_\kappa$ is an (eventually ordered[10]) collection of signatures of morphisms whose codomains correspond to properties forming the *primary identifier* of kind κ.
- ref_κ is a *set of references* from κ, i.e., a set of pairs $(name_{\kappa'}, R_{\kappa'})$, where $name_{\kappa'}$ is the name of the referenced kind κ' and $R_{\kappa'}$ is a set of referenced properties of kind κ'. It must hold that access path P_κ contains mapping of properties in $R_{\kappa'}$ to enable reconstruction of the relationship between referring and referenced properties of both κ and κ'.
- P_κ is an *access path*, i.e., a description of the internal structure of κ.

In the case of references, there can occur three cases:

1 *null*: The model does not support references at all (but we can still keep them in the categorical framework and check externally).
2 $\emptyset$: The model supports references, but none of them is used.
3 The set has at least one input, because there is at least one reference in the model.

[10] If required by the respective model.

Example 3.3 In the case of the relational model, e.g., in PostgreSQL, examples of $pkey_\kappa$ can be {1}, (1), or {3, 5}. In the case of *Cassandra* $pkey_\kappa$ can be ((3.21, 5.21), 25.23) since the system allows the grouping of parts of the key. □

Access path P_κ is represented as a tree, where each node corresponds to one property of kind κ and the edges represent the mutual nesting of properties if supported by the respective model. Furthermore, the sibling properties may be ordered in some models. The root of the tree is an auxiliary node, its child nodes correspond to top-level properties of κ. Each node is simultaneously a root of an access subpath, describing the structure of the respective nested property.

Each node (property) ϕ of the tree is represented as a tuple $(name_\phi, context_\phi, value_\phi)$. In the case of the auxiliary root node $name_\phi = \epsilon$, $context_\phi = null$, and $value_\phi$ represents the structure consisting of top-level properties of κ. If property ϕ' is the parent of property ϕ, there exists a (base/composite) morphism $m_{child} : o_{\phi'} \rightarrow o_\phi \in \mathcal{M}_\mathbf{S}$, where $o_\phi, o_{\phi'} \in \mathcal{O}_\mathbf{S}$ are objects representing properties ϕ, ϕ'.

$name_\phi$ represents the *name* of property ϕ and can be of the following types:

- A *static name* corresponding to a fixed value, either *inherited* from schema category or *user-defined*.
- An *anonymous* (empty) name.
- A *dynamically derived name* corresponding to signature of (base/composite) morphism $m_{name} : o_\phi \rightarrow o_{name}$, where o_{name} is the object representing dynamically derived names. (Cardinality of the respective base morphism(s) must be (1, 1).)

In addition, there are specific features of the properties of particular data models that need to be reflected too: First, since the XML document model allows two kinds of properties, i.e., an XML element and an XML attribute, we distinguish between them using the prefix @ used for attributes. Second, edges in the graph model are mapped using properties with pre-defined (reserved) names *_src* for the source and *_tgt* for the target of the edge, respectively.

Optional $context_\phi$ represents the *context* of property ϕ within parent property ϕ', i.e., it denotes the root object o_ϕ, if any, associated with ϕ. We distinguish the following cases:

- If $context_\phi$ is a signature of base morphism m_{child}, it represents the case when o_ϕ is a direct neighbour of $o_{\phi'}$ in the graph of $\mathbf{S}$.
- If $context_\phi$ is a signature of composite morphism m_{child}, it represents *inlining* of property ϕ to ϕ' from a more distant position in the graph of $\mathbf{S}$.
- If $context_\phi$ is undefined, there exists no $o_\phi \in \mathcal{O}_\mathbf{S}$, but its content ($value_\phi$) does exist. It corresponds to the case when the property ϕ is a simple property (i.e., a leaf of the access path) or when the user adds an *auxiliary property* ϕ, e.g., to group a set of selected related properties.

Finally, $value_\phi$ represents the particular (simple or complex) *value* of property ϕ. We distinguish three cases:

```
STRUCTURE -> { (NAME : CONTEXT VALUE (, NAME : CONTEXT VALUE)*)? }
NAME      -> ( static-name | _ | SIGNATURE )
CONTEXT   -> ( SIGNATURE )?
VALUE     -> ( SIGNATURE | ARRAY | STRUCTURE | epsilon )
ARRAY     -> [ ((NAME :)? CONTEXT VALUE (, (NAME :)? CONTEXT VALUE)*)? ]
SIGNATURE -> m_id(.m_id)*
```

Fig. 5 Grammar of the JSON-like representation of access paths

- A *simple value* is a signature of morphism $m_{value} : o_\phi \rightarrow o_{value}$, where o_{value} is the object representing the simple values.
- An *array* is as an ordered list of recursively defined nested properties $\phi_1, ..., \phi_l$, for some $l \in \mathbb{N}^+$.
- A nested *structure* is an unordered set of recursively defined nested properties $\phi_1, ..., \phi_k$, for some $k \in \mathbb{N}^+$.

The latter two are denoted as a *complex value* of a property.

JSON-like representation

For the sake of easier processing and understanding, we introduce a textual JSON-like representation of an access path. It is defined by the grammar depicted in Fig. 5. `STRUCTURE` is the start nonterminal. Terminal `static-name` represents an inherited or user-defined name of a property. Terminal `_` (underscore) represents the anonymous name of a property. Terminal `epsilon` represents an empty value. Terminal `m_id` represents the *signature* of a base morphism and terminal `.` (dot) represents their concatenation. Terminals `{` and `}` (curly brackets), `[` and `]` (square brackets), `,` (comma), and `:` (colon) serve as delimiters.

As we can see, there are three positions, where the signatures of morphisms (`SIGNATURE`) occur—in the case of dynamically derived names, specification of the context of a property, and specification of simple value of a property. Adding another level of nesting of properties (`STRUCTURE`) can occur at two positions—as a complex value of a property or as an element of an array. (Also note that for simplicity we do not consider data types. This information could however be added between `CONTEXT` and `VALUE` as a system-specific `TYPE`.)

Example 3.4 Fig. 6 illustrates the access path of collection *Order* from Fig. 1. We remind the collection itself on the left and the respective part of schema category **S** on the right. In the middle we can see the respective access path. The colors denote the corresponding parts in all three data representations.

As we can see in the figure, the description starts from the value part of the auxiliary root node, i.e., its empty name and context are omitted. It consists of top-level properties *_id*, *contact*, and *items*.

The first one corresponds to a nested document having an auxiliary user-defined name *_id* that is not present in schema category **S** and two nested (leaf) properties *customer* and *number*.

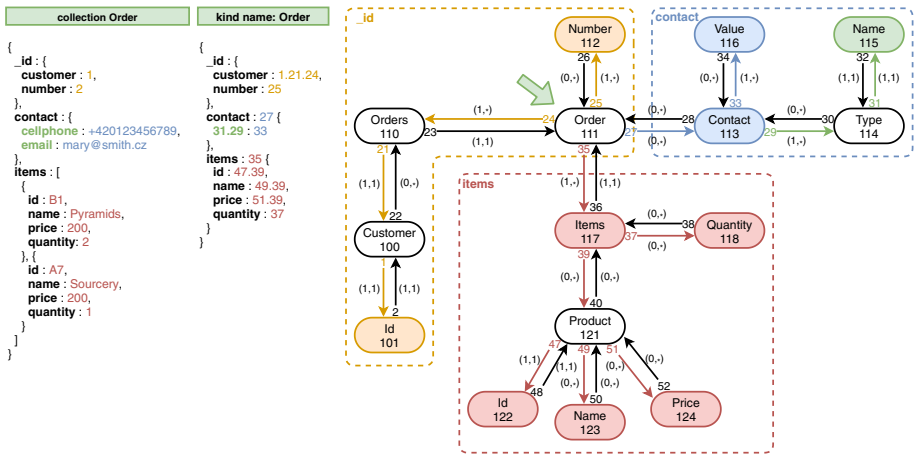


Fig. 6 Collection *Order*, an access path for kind *Order*, and the corresponding part of schema category **S**

The second one is a map *contact* having the context specified by a morphism with *signature* $= 27$ and containing a set of pairs (name : value) distinguishable using dynamically derived names and corresponding values. Note, that the corresponding object from schema category **S** has *superid* $= \{31.29, 33\}$, making them related.

The third one is a homogeneous array *items* of an anonymous complex type. Note that the cardinality of morphism determines the fact that it is an array with *signature* $= 35$. The anonymous nested complex property (document) corresponds to a set of four properties. Properties *id*, *name*, and *price* are related to object *Product*. Property *quantity* is related to object *Items*. In other words, the mapping allows collocating properties that are not directly mutually related in the schema category. □

Note that there is a difference between aggregate-ignorant and aggregate-oriented models. For aggregate-ignorant ones, there is no need to consider `ARRAY` or `STRUCTURE` in `VALUE`. Moreover, specifying of `CONTEXT` is mandatory. In the relational model, only morphisms having cardinality (0, 1) or (1, 1) are allowed to connect a property (i.e., attribute) with a kind (i.e., relational table). In addition, the graph model allows a homogeneous array of a simple type, i.e., other cardinalities are allowed.

Aggregate-oriented models allow more complex structures. Among others, the following commonly used data structures that the grammar can describe are thus supported:

- *Heterogeneous array*: In the most general situation, a heterogeneous array can have a morphism specifying its context with cardinality $(0, N)$, or it has no morphism. Items of the array also can have the morphism specifying their context within the array with cardinality $(0, N)$. In general, any cardinalities which allow an array of at least two items with distinct types are allowed.

- *Tuple*: A tuple is a special kind of a heterogeneous array. The morphism specifying the context of a tuple has an arbitrary cardinality, or it has no specifying morphism. Items of the tuple have the morphism specifying their context within the tuple with cardinality (1, 1) (or even (0, 1) in case, e.g., *Cassandra*).
- *Nested document*: In the case of nested documents, the same rules are applied as to the top-level document.
- *Map*: A map is a special kind of nested document, having dynamically derived names of properties.

Transformations

Having defined the schema category $\mathbf{S}$, instance category $\mathbf{I}$, and mapping $\mathfrak{M}$ between the categorical representation and particular models, in this section, we can introduce the algorithms for mutual transformation between categorical and logical data representations. We aim to provide a generic approach applicable to all data models (and their combinations). After we define the transformation process for both directions, we discuss how it can be used, e.g., for data migration.

Model-to-category transformation

First of all, we describe the process of data transformation from a particular logical model to the categorical representation. It consists of two steps: 1) we fetch data from an input logical model and 2) we insert selected records, one-by-one, to instance category $\mathbf{I}$.

Forest of records

To be able to uniformly manipulate records from different data models (recall Table 1), both aggregate-oriented and aggregate-ignorant, we first propose their tree-based representation. Each record r is represented as a directed (eventually ordered[11]) tree $r = (V, E)$. V contains a node v_i for each (eventually nested) property ϕ_i, $i = 1, \ldots, n$, in record r (only if property ϕ_i appears in access path as a mapping of a categorical object) and an auxiliary root node $v_0 \in V$ representing the whole record denoted as ϕ_0. Each node $v \in V$ contains an array of name/value pairs $(name_v, value_v)$, where $name_v$ represents the name of the property and $value_v$ represents its value. Nodes $v_j, v_k \in V$ are connected using a directed edge $e = (v_j, v_k)$, $e \in E$ if the corresponding properties ϕ_j, ϕ_k in record r are in a parent/child relationship, i.e., property ϕ_k is nested in property ϕ_j. Hence, a property with a simple type or a property representing an array of a simple type is represented as a leaf node, while other types of properties are represented as an inner node.

Records of the same kind κ are grouped to form a *forest of records* $F_\kappa = (T_\kappa, M_\kappa)$, where T_κ is a set of trees representing the records of κ and M_κ is a mapping that maps a *categorical identifier* of each property ϕ occurring in kind κ to the list of the respective nodes in trees in T_κ. The categorical identifiers correspond to a pair $name_\phi : context_\phi$ for inner nodes and $name_\phi : value_\phi$ for leaf nodes. The mapping

[11] Depending on the particular model.

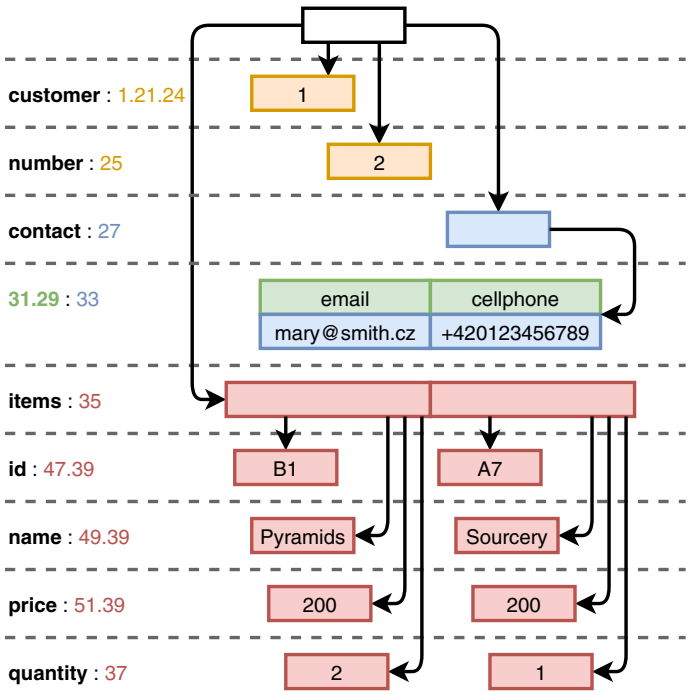


Fig. 7 Forest containing a single record corresponding to a document

allows a quick access to all properties corresponding to the same instance category object at the same level of trees in T_κ. Hence, there is no need to traverse the whole tree to access a particular property. (Note that we do not materialize the whole forest for all input trees. Only the currently processed data fragments are constructed for further processing.)

Example 4.1 Fig. 7 illustrates the representation of document *Order* corresponding to the access path depicted in Fig. 6 as a tree (in a forest of size 1). On the left we can see the categorical identifiers, on the right the particular tree, whereas the levels represent the mapping. (Note that to simplify the figure, we do not depict $value_v$ of node v if $name_v$ is user-defined and thus it is a part of the categorical identifier.) The root of the tree corresponds to the document itself. All leaves correspond to properties with a simple type or an array of a simple type. Other nodes represent more complex structures. For example, node *items* corresponds to a complex-type array. Anonymous node _ corresponds to a nested document. Node *contact* corresponds to the map of contacts having dynamically derived names of properties. (Note that node *_id* does not appear in the forest of records, since the corresponding object is not in the schema category). □

Example 4.2 As illustrated in Fig. 9, the representation of records in relational table *Customer* is significantly simpler, since there are no hierarchical structures. In the figure,

we can see the mapping of each categorical identifier (on the left) to all respective properties in all trees depicted at the same level (on the right). □

Transformation algorithm

The input of the algorithm is formed of schema category **S**, (possibly non-empty) instance category **I** corresponding to **S**, the forest of input records $F_\kappa = (T_\kappa, M_\kappa)$ of kind κ, access path P_κ of kind κ, root object $root_\kappa$ and root morphism $morph_\kappa$ associated with κ. A model-specific command creates the forest of records (expressed in pseudocode, e.g., like `SELECT * FROM KIND` κ), followed by model-specific transformation of its result to the forest structure F_κ. In "Framework MM-cat" section we show the respective implementation for particular models using wrappers.

The algorithm processes one-by-one every input record (tree) $r \in T_\kappa$. Based on the DFS traversal, it traverses the access path P_κ which describes the required mapping and fills instance category **I** with appropriate data fragments. The pseudocode of the transformation algorithm is provided in Algorithm 1.

As we can see, processing one record r consists of two phases—preparation and processing of the rest of the tree.

Preparation Phase In the preparation phase, we distinguish two situations—if kind κ is associated with a root object or a root morphism. In the former case (line 8), we first gain object $q_\mathbf{I}$ corresponding to $root_\kappa$ using functor $Inst_\mathbf{I} : \mathbf{S} \to \mathbf{I}$. Next, using function *fetchSids()* we acquire a set S which consists of sets of pairs (*name*, *value*), where *name* corresponds to a particular *superid* attribute of $root_\kappa$ and *value* corresponds to the respective value in r, if it exists. (In the case of $root_\kappa$, every record r is identified using a single (super)identifier, i.e., $|S| = 1$.) Note that we work with the keys of schema category objects used both in the access path P_κ and in the mapping F_κ used in the input forest of records.

Example 4.3 Consider again Figs. 7 and 6. Object o_κ corresponding to *Order* is identified by a $superid = \{1.21.24, 25\}$ corresponding to objects *Id* (with $key = 101$) and *Number* (with $key = 112$). Function *fetchSids()* exploits mapping M_κ to quickly navigate to specific values of properties *customer* and *number*, matches them to corresponding keys of objects representing these properties in **S** and returns set S that contains a single set $\{(1.21.24, 1), (25, 2)\}$. □

Then, the algorithm iterates through the set S. Each $sid \in S$ internally modifies object $q_\mathbf{I}$ and participates in further traversing of access path P_κ. Internal modification of $q_\mathbf{I}$ is done in function *modifyActiveDomain()* (line 12), where four cases may occur:

- If $sid \in q_\mathbf{I}$, nothing has to be done.
- If sid is a part of an already existing $sid_\mathbf{I} \in q_\mathbf{I}$, sid is replaced by $sid_\mathbf{I}$.
- If sid corresponds to an already existing $S_\mathbf{I} \subseteq q_\mathbf{I}$, sid replaces $S_\mathbf{I}$.

- If $sid \notin q_{\mathbf{I}}$, it is added.

Further traversing is ensured by function *children()* (line 13) which determines the new context and value to be processed in the same way. (We describe its body in detail in paragraph *Function children()* on page 21.) The result of the function associated with a particular *sid* is then pushed to the top of auxiliary stack M as a triple (*sid*, *context*, *value*). The reason for also involving *sid* is that we need to know the associated parent in the next steps to appropriately fill the morphisms *context* between corresponding parent and child objects in $\mathbf{I}$.

In the second option, i.e., if κ is associated with a root morphism (line 15), we gain both the domain and codomain of the root morphism $morph_{\kappa}$. Next, for both of them, we also fetch the sets of corresponding superidentifiers using function *fetchSids()* and we apply function *modifyActiveDomain()* respectively. In lines 22 and 23 we fill relations corresponding to the root morphism and its dual morphism. Using function *getSubpathBySignature()* we get an access subpath t' of access path t provided in the first parameter corresponding to the signature of morphism m provided in the second parameter. In particular, it is a subpath t' such that every leaf l of t' has $l.context = m$ or $l.value = m$ or any ancestor a of l has $a.context = m$. If there are more such subpaths, the one closest to t is returned. If m is *null*, then l such that $l.value = \epsilon$ is returned.

Finally, we acquire all new pairs (*context*, *value*) to be processed regarding the root morphism's domain and codomain to ensure further traversing. These pairs, except for the one representing the already processed root morphism, are then pushed to the auxiliary stack M together with respective *sid*s.

Processing of the Tree After having completed the initial phase, the algorithm one-by-one releases and processes the top of the stack M until it is empty. The released triple $(pid, m_{\mathbf{S}}, t)$ forms the new context of the algorithm, i.e., context morphism $m_{\mathbf{S}}$ and access (sub)path t associated with parent superidentifier *pid*. Morphism $m_{\mathbf{I}} : p_{\mathbf{I}} \rightarrow o_{\mathbf{I}}$ and object $q_{\mathbf{I}}$ are then computed using functor $Inst_{\mathbf{I}}$ (line 32 and 34).

Once again, we fetch S as a set of superidentifiers corresponding to $o_{\mathbf{S}}$ (being codomain of $m_{\mathbf{S}}$) from record r associated with currently processed *pid* (i.e., there is an edge $(pid, sid) \in r$). This time size of S is not limited by 1 since the cardinalities of the properties allow multiplicity. S being fetched, the algorithm iterates through $sid \in S$ and processes each of them in order:

1 to internally modify the active domain of object $q_{\mathbf{I}}$ (line 37),
2 to add relations for $m_{\mathbf{I}}$ (lines 38, 39), and
3 to participate in the further traversing of access path t (lines 40, 41).

Note that function *fetchSids()* returns only superid sets that are constructed from properties having as an ancestor value *pid* in the currently processed record r. In the preparation phase, the same function returns superid values related to *null*, e.g., having no ancestor.

Also note that the function *fetchSids()* returns an empty set if the data corresponding to the fragment of the access path does not occur in the record. As a consequence of an empty set of sids, the (possible) traversing of corresponding access subpath stops, since there is no data in the record to be traversed (applies for both simple and complex properties).

As for adding of relations, we distinguish two situations. If $m_{\mathbf{I}}$ is a base morphism, we only add pair (*pid*, *sid*) to morphism $m_{\mathbf{I}}$ and mapping (*sid*, *pid*) to dual morphism $m_{\mathbf{I}}^{-1}$. If $m_{\mathbf{I}}$ is a composite morphism, we add relations to all base morphisms forming the composite morphism $m_{\mathbf{I}}$. Thus we need to extend also the active domains of the affected objects, respectively. To do so, the algorithm either determines the superidentifier of such objects from *r*, or computes a technical identifier (i.e., autoincrement).

The algorithm ends when the stack *M* is empty meaning that all the data are transformed into instance category **I**, i.e., internal structures of objects and morphisms in **I** are appropriately extended.

Example 4.4 Suppose that we have an access path depicted in Fig. 6 and a corresponding forest of records depicted in Fig. 7. The intended transformation should convert the data represented in the document model to the categorical representation corresponding to the schema category **S** depicted in Fig. 3 and non-empty instance category **I**.

The algorithm processes each record *r* as follows: First, properties *customer* and *number* corresponding to the superidentifier of object *Order* are fetched from record *r*, and as a set of tuples, i.e., $\{(1.21.24, 1), (25, 2)\}$, added to set *S* as the document identifier, i.e., a part of the superidentifier of object *Order* from schema category **S**. Next, instance category **I** is extended using *sid*, i.e., the active domain of corresponding object $q_{\mathbf{I}}$ is extended with the value of *sid*. And access path P_K (depicted in Fig. 6) is traversed, creating triples for stack *M* for property *Customer*, *Number*, *Items*, and *Contact* related to *sid*, as depicted in Fig. 10. In the figure on the right we can also see the current content of instance category **I**, i.e., a particular order was added.

Next, the top of the stack is released, i.e., the triple describing the access subpath leading to property *items*, i.e.:

```
{ id : 47.39,
  name : 49.39,
  price : 51.39,
  quantity : 37 }
```

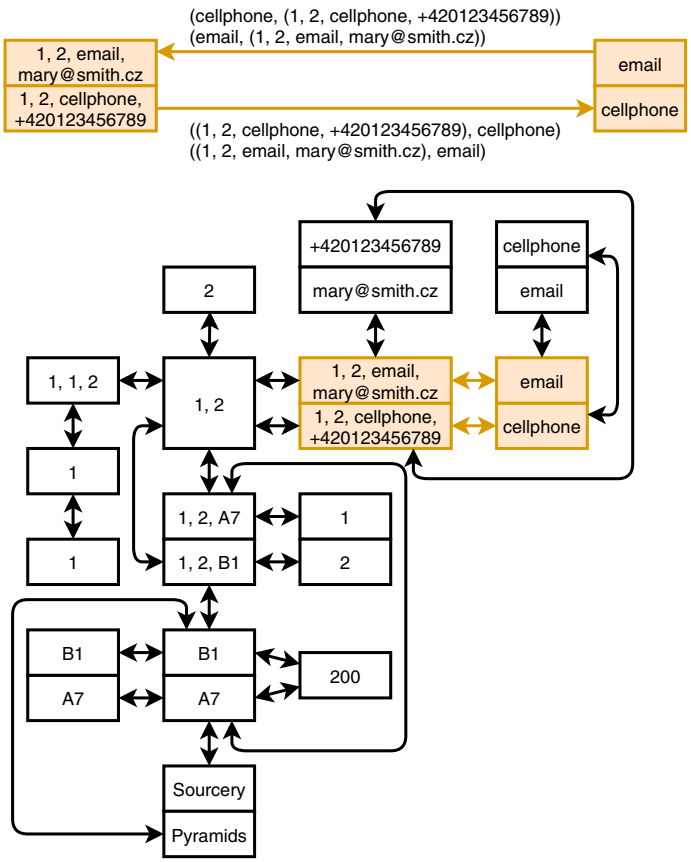


Fig. 8 Part of instance category **I** after the insertion of single Order

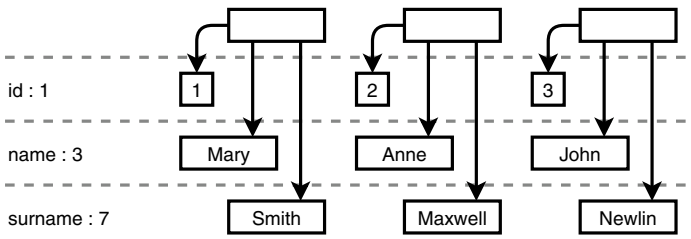


Fig. 9 Forest containing three records corresponding to a row of a relational table

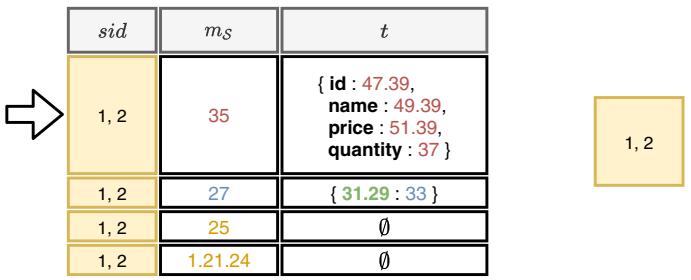


Fig. 10 Stack *M* and instance category **I** when the preparation phase for data from Fig. 7 is completed

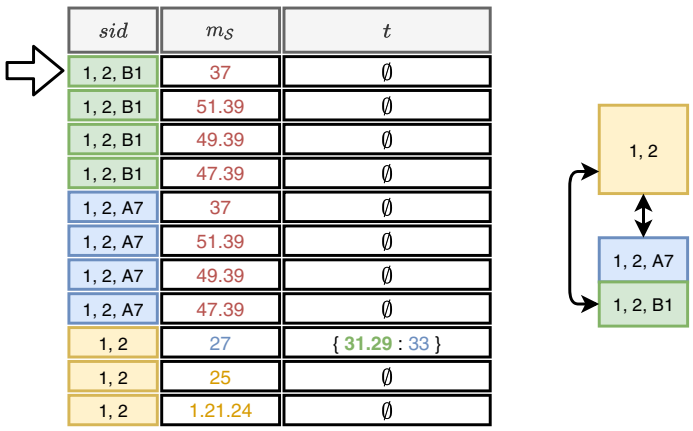


Fig. 11 Stack *M* and instance category **I** when the first iteration of tree processing completed

associated with pid (*customer* : 2, *number* : 1) and morphism $35_{\mathbf{I}} : Order_{\mathbf{I}} \rightarrow Items_{\mathbf{I}}$ of instance category **I**. Within this context, the active domain of object $Items_{\mathbf{I}}$ is filled with the following tuple:
$\{(1.21.24.36, 1), (25.36, 2), (47.39, \text{“}A7\text{”})\}$

Relation:

$(\{(1.21.24.36, 1), (25.36, 2), (47.39, \text{“}A7\text{”})\},$

$\{(1.21.24, 1), (25, 2)\})$

is added to morphism $m_{\mathbf{I}}$ and dual morphism $m_{\mathbf{I}}^{-1}$ is extended with relation:

$(\{(1.21.24, 1), (25, 2)\},$

$\{(1.21.24.36, 1), (25.36, 2), (47.39, \text{“}A7\text{”})\})$

Finally, the access path leading to property *items* is further traversed to the access paths corresponding to leaves, i.e., 47.39, 49.39, 51.39, and 37.

The same applies to the other *sid*, i.e.,

$(1.21.24.36, 1), (25.36, 2), (47.39, \text{“}B1\text{”})$

as can be seen in Fig. 11. The algorithm continues in the same way until stack *M* is empty. The resulting part of the instance category **I** corresponding to kind *Order* is depicted in Fig. 8. □

Algorithm 1: Model-to-Category Transformation

```
Input: S – schema category
       I – instance category
       F_κ = (T_κ, M_κ) – forest of input records
       P_κ – access path associated with κ
       root_κ – root object associated with κ
       morph_κ – root morphism associated with κ
M ← an empty stack
foreach record r in T_κ do
    // preparation phase:
    if morph_κ is null then
        // κ with root object:
        q_I := Inst_I(root_κ)
        S := fetchSids(root_κ.superid, r, null)
        foreach sid in S do
            sid := modifyActiveDomain(q_I, sid)
            foreach (context, value) in children(P_κ) do
                M.push((sid, context, value))
    else
        // κ with root morphism:
        S_dom := fetchSids(root_κ.superid, r, null)
        sid_dom := modifyActiveDomain(Inst_I(root_κ), S_dom.get(0))
        q_cod := morph_κ.cod
        S_cod := fetchSids(q_cod.superid, r, null)
        sid_cod := modifyActiveDomain(Inst_I(q_cod), S_cod.get(0))
        m_I := Inst_I(morph_κ)
        addRelation(m_I, sid_dom, sid_cod, r)
        addRelation(m_I^-1, sid_cod, sid_dom, r)
        t_dom := getSubpathBySignature(P_κ, null)
        t_cod := getSubpathBySignature(P_κ, morph_κ)
        foreach (context, value) in (children(P_κ) \{(t_dom, t_cod)}) do
            M.push((sid_dom, context, value))
        foreach (context, value) in children(t_cod) do
            M.push((sid_cod, context, value))
    // processing of the tree:
    while M is not empty do
        (pid, m_S, t) := M.pop()
        m_I := Inst_I(m_S)
        o_S := m_S.cod
        q_I := Inst_I(o_S)
        S := fetchSids(o_S.superid, r, pid)
        foreach sid in S do
            sid := modifyActiveDomain(q_I, sid)
            addRelation(m_I,pid, sid, r)
            addRelation(m_I^-1,sid, pid, r)
            foreach (context, value) in children(t) do
                M.push((sid, context, value))
```

Function children() Having the whole algorithm built on the DFS principle, the main purpose of function *children()* is to determine the access subpaths to be traversed from the input access path *t*. The function (see Algorithm 2) returns a set *C* of pairs (*context*, *value*), each consisting of possibly non-empty access sub-path *value* and morphism *context*, both corresponding to currently traversed access path *t*.

Algorithm 2: Function *children()*

```
1  function children(t):
       Input: t – access (sub)path
2      C := ∅
3      foreach top-level property p in t do
4          C.addAll(traverseAccessPath(p.name, ∅, ∅))
5          C.addAll(traverseAccessPath(∅, p.context, p.value))
6      return C
```

For each top-level property of access path *t* modeled as a triple (*name, context, value*) we traverse its *name* separately. Its *context* and *value* are traversed together to determine the body of the property. Both cases are ensured by calling function *traverseAccessPath()*—see Algorithm 3. While the *context* may contain a base/composite morphism, the *value* may contain a base/composite morphism or a complex structure. As we can see in the algorithm, multiple cases may occur:

- If a *name* is static or anonymous, nothing has to be done. There is nothing to traverse, so an empty set is returned.
- If a *name* is a signature of a base/composite morphism, its dynamic name must be computed and further traversed. Thus, the *name* and an empty access sub path are added (corresponding to the fact that it represents a leaf).
- If the *value* is a signature or empty, i.e., a simple value, the concatenation of *context* and *value* is returned together with an empty set to be further traversed.
- If the *context* is a signature and the *value* is complex, the pair (*context, value*) is returned.
- Else, i.e., if there is no specified context, we must further traverse *value* to determine *context*. Hence, the function *children*() is recursively called.

Algorithm 3: Function *traverseAccessPath()*

```
1  function traverseAccessPath(name, context, value):
       Input: name – the name of a particular property (static, anonymous, dynamic)
2             context – optional context of a property (allowing, e.g., grouping)
3             value – simple or complex value of a property
       // static name:
4      if name is static_name or name is _ then
5          return ∅
       // dynamic name:
6      else if name is SIGNATURE then
7          return {(name, ∅)}
       // simple value possibly with a context:
8      else if value is SIGNATURE or value is ϵ then
9          return {( context++value, ∅)}
       // complex value having context:
10     else if context is SIGNATURE then
11         return {(context, value)}
       // complex value without context:
12     else
13         return children(value)
```

Category-to-model transformation

Having an instance category $\mathbf{I}$ and mapping $\mathfrak{M}$, the opposite direction of transformation allows extraction of data from $\mathbf{I}$ and storing it into a particular logical model. The whole algorithm consists of three parts:

1. *DDL Algorithm*: Definition of the schema of the data including names of properties that are dynamically derived (see "DL algorithm" section).
2. *DML Algorithm*: Transformation of data instances from instance category $\mathbf{I}$ to a particular logical model (see "DML algorithm" section).
3. *IC Algorithm*: Finalization of schema definition with integrity constraints, i.e., adding of identifiers and references to other kinds (see "IC algorithm" section).

DDL algorithm

Having a schema category $\mathbf{S}$, instance category $\mathbf{I}$, access path P_κ, kind name $name_\kappa$, and particular database wrapper W_D working over database D, the first algorithm creates a DDL statement to define a schema of kind κ in database D, i.e., a statement of type `CREATE KIND`. The algorithm proceeds "lazily". First, it provides all the information about the structure of the currently processed kind κ to wrapper W_D. Second, it calls the method for constructing the output database-specific command. The command can be sent to D for execution or just visualized to the user, e.g., for checking.

The processing is again based on the DFS approach. The traversal of P_κ is implemented using stack M that contains the context of the traversing (N_p, t), i.e., set of names N_p that correspond to the property represented by access sub-path t. There can be more than one name in N_p if the property's name is dynamically derived. In addition, since the structure of κ can be hierarchical, for easier construction of the resulting command, the names in the context are constructed using their concatenation expressing the path from the root of the hierarchy (e.g., `/Order/Items/_/Name`)—we denote them as *hierarchical names*.

As we can see in Algorithm 4, we begin the processing with the setting of kind name $name_\kappa$ to wrapper W_D and we check whether the schema is applicable, i.e., whether database D is not schema-less. If D is schema-less, only a trivial DDL statement is returned, i.e., kind κ is created without specification of its structure (for example, in *MongoDB* this would be command `db.createCollection("orders")`). Otherwise, traversing of the access path P_κ is carried out using stack M. It is initialized by pushing the initial context, i.e., set N_0 containing only trivial name ϵ (since the whole kind κ does not have a parent name) and the whole access path P_κ associated with kind κ.

We iterate through the body of while cycle until the stack M is empty. First, we release from the top of the stack M the currently processed context (N_p, t), i.e. a set of hierarchical property names N_p corresponding to parent property p of the property represented by access sub-path t. Next, using function *determinePropertyName()* we construct the set of names N_t of the current property. And we construct the set of new hierarchical names N as a concatenation of pairs resulting from Cartesian product $N_p \times N_t$.

Depending on whether t describes a simple property (i.e., *t.value* corresponds to a `SIGNATURE` or it is empty) or a complex property we add new properties to wrapper W_D. If t describes a simple property (line 14), we create a new property for each name $n \in N$ within kind κ.[12] Exploiting the cardinalities in schema category $\mathbf{S}$, we further specify whether the new property is an array or optional. If t describes a complex property (line 21), the processing is similar, but the wrapper is informed about a complex property or an array of complex properties. In addition, we push all child properties to stack M (line 28) to be processed as well.

Finally, using the wrapper W_D the algorithm constructs and returns the particular DDL statement. If D already contains a kind of the same name, the statement can be of type `ALTER KIND`, otherwise statement of type `CREATE KIND` is created.

Function determinePropertyName() This function returns the resulting name (or a set of names) depending on the way it was specified by the user. If the name is statically determined (user-defined, anonymous, or inherited from schema category $\mathbf{S}$), it directly forms the output of the function. If the name is dynamically derived, the function acquires all values stored in the active domain of the object specified using a signature of its input morphism. The set of values forms the output of the function.

Algorithm 4: DDL Algorithm

```
   Input: S – schema category
1          I – instance category
2          name_κ – name of kind κ
3          P_κ – access path associated with κ
4          W_D – DDL wrapper for a database D
5  N_0 ← {ϵ}
6  M ← an empty stack
7  W_D.setKindName(name_κ)
8  if W_D.isSchemaLess() is False then
9      M.push((N_0, P_κ))
10     while M is not empty do
11         (N_p, t) := M.pop()
12         N_t := determinePropertyName(I, t.name)
13         N := concat(N_p × N_t)
14         if t.value is SIGNATURE or t.value is ϵ then
               // processing of a simple property:
15             opt := isOptional(min(S, t.context++t.value))
16             array := isArray(max(S, t.context++t.value))
17             if array is True then
18                 W_D.addSimpleArrayProperty(N, opt)
19             else
20                 W_D.addSimpleProperty(N, opt)
21         else
               // processing of a complex property:
22             opt := isOptional(min(S, t.context))
23             array := isArray(max(S, t.context))
24             if array is True then
25                 W_D.addComplexArrayProperty(N, opt)
26             else
27                 W_D.addComplexProperty(N, opt)
28             foreach triple c in t do
29                 M.push((N, c))
30 return W_D.createDDLStatement()
```

[12] If there are multiple names in *N*, their processing can in some systems differ. For example, while the wrapper for *PostgreSQL* would create separate properties, the wrapper for *Cassandra* would create a map of properties.

DML algorithm

Having the schema category **S**, instance category **I**, kind name $name_\kappa$, access path P_κ, root object $root_\kappa$ and root morphism $morph_\kappa$, both associated with kind κ, and particular database wrapper W_D working over database D, the second algorithm creates a list of DML statements which store data into the schema of kind κ in database D, i.e., statements of type `INSERT INTO KIND`. If the resulting commands are sent for execution to database D, they can fill in the kind created using Algorithm 4 with data from instance category **I**.

As we can see in Algorithm 5, we first initialize an empty name $n_0 = \epsilon$, empty list *dml*, and empty stack M. The rest of the processing depends on whether κ has a root object or a root morphism. In the former case, we first acquire object $q_\mathbf{I} \in \mathbf{I}$ corresponding to $root_\kappa$ using functor $Inst_\mathbf{I}$. In the next step, we get the active domain S of $q_\mathbf{I}$. We push each $sid \in S$ together with empty name n_0 and P_κ to auxiliary stack M and we call function *buidStatement()* (see below) which creates the respective `INSERT` command that is then added to list *dml*.

In the latter case, i.e., κ with the root morphism, we first acquire the respective morphism $m_\mathbf{I}$ using the functor $Inst_\mathbf{I}$. Next, using function *fetchRelations()* we get a set of all pairs (o_1, o_2), where $o_1, o_2 \in \mathcal{O}_\mathbf{I}$ such that $m_\mathbf{I}(o_1) = o_2$. Then we get access subpath t_{cod} of codomain $morph_\kappa.cod$ using function *getSubpathBySignature()*. For each $s \in S$ we initialize stack M with two values—one for the domain (line 22) and one for the codomain (line 23). In the former case, we use the original access path P_κ without subpath t_{cod} corresponding to the codomain. In the latter case we use the so-far unprocessed subpath t_{cod}. Then we call function *buildStatement()* and add its result to the list *dml*.

Function buildStatement() As stated in Algorithm 6, function *buildStatement()* iteratively processes the initialized stack M until it is empty. First, the top of M is released as a triple consisting of an identifier of parent property *pid*, hierarchical property name n_p, and respective access (sub)path t. Using function *collectNameValuePairs()* we acquire a set of pairs (*name*, *value*)[13] of data from **I** relative to *pid* as specified by t. Each pair (*name*, *value*) is then processed as follows: If t describes a simple property (line 11), the algorithm calls the wrapper to extend the current `INSERT` statement by adding *value* to kind κ as an attribute named $n_p{+}{+}name$. It is up to the wrapper to determine how the empty data (null) will be inserted. It is a model-dependent feature if the missing data leads to a missing property or a *null* metavalue. If t describes a complex property (line 15), the algorithm iterates through the set of nested properties within the complex property and for every such property it pushes to stack M the respective new triple, i.e., it moves the processing to the next level. After processing of whole stack M, the wrapper is invoked to create and return the final `INSERT` statement.

[13] Note that if we acquire multiple values by traversing a morphism having the upper bound cardinality set to many (i.e., $*$), the *name* part is distinguished by an index in a form *name*[i], $i \in \mathbb{N}$. Also note that if the *value* for a particular *name* is missing (in the categorical approach we represent missing values (`null`) as a missing relation in a morphism), the resulting pair contains *value* set to ϵ.

Algorithm 5: DML Algorithm

```
   Input: S – schema category
1          I – instance category
2          name_κ – name of kind κ
3          P_κ – access path associated with κ
4          root_κ – root object associated with κ
5          morph_κ – root morphism associated with κ
6          W_D – DML wrapper for database D
7  n_0 ← ϵ
8  dml ← an empty list
9  M ← an empty stack
10 if morph_κ is null then
       // κ with root object:
11     q_I := Inst_I(root_κ)
12     S := fetchSids(q_I)
13     foreach sid in S do
14         M.push((sid, n_0, P_κ))
15         stmt := buildStatement(W_D, I, M, name_κ)
16         dml.add(stmt)
17 else
       // κ with root morphism:
18     m_I := Inst_I(morph_κ)
19     S := fetchRelations(m_I)
20     t_cod := getSubpathBySignature(P_κ, morph_κ)
21     foreach (o_1, o_2) in S do
22         M.push(o_1, n_0, P_κ.minusSubtree(t_cod))
23         M.push(o_2, n_0, t_cod))
24         stmt := buildStatement(W_D, I, M, name_κ)
25         dml.add(stmt)
26 return dml
```

Algorithm 6: Function *buildStatement()*

```
1  function buildStatement(W_D, I, M, name_κ):
       Input: W_D – DML wrapper for a database D
2             I – instance category
3             M – context stack
4             name_κ – name of kind κ
5      W_D.clear()
6      W_D.setKindName(name_κ)
7      while M is not empty do
8          (pid, n_p, t) := M.pop()
9          P := collectNameValuePairs(I, t, pid)
10         foreach (name, value) in P do
11             if t.value is SIGNATURE or t.value is ϵ then
                   // processing of simple property:
12                 W_D.append(n_p++name, value)
13             else if value is ϵ then
                   // processing of empty complex property:
14                 W_D.append(n_p++name, value)
15             else
                   // processing of non-empty complex property:
16                 foreach top-level subtree t' in t do
17                     M.push((value, n_p++name, t'))
18     return W_D.createDMLStatement()
```

IC algorithm

This algorithm aims to modify the created kinds to add integrity constraints ensuring the respective identifiers and references. These parts of schema definition are the most system-specific ones; however, the proposed approach is general enough to cover all known cases. The intra-model references are propagated to the respective DBMS by the system-specific wrapper. In the case of inter-model references, the propagation differs depending on the underlying combination of systems. A traditional polystore, as well as a multi-model DBMS is considered a separate system having its single wrapper, so the system itself handles the inter-model reference. In the case of a polystore-like combination of systems, where each has its wrapper, the DBMSs (naturally) cannot handle the references, because they are not aware of each other. However, the proposed categorical framework keeps this information and, thus, the integrity constraints can be checked externally. And, in general, this external checking of integrity constrains can also be used for a single-model DBMS lacking a support for references.

The whole process is described in Algorithm 7 which extracts all primary identifiers and references related to a particular mapping m and ensures their application at the logical level of D using a command of type `ALTER KIND`. Its input is formed of mapping $m \in \mathfrak{M}$, and respective wrapper W_D. First, we process the identifier of κ. Using function *collectNames()* we get ordered collection N which contains attributes of the identifier of κ. The result is added to the wrapper W_D for system-specific processing. Note that we use names from $m.P_\kappa$ which are, contrary to user-defined names, unique.

Next, we process the set of references by iterating through set $m.ref_\kappa$. First, using function *collectSigNamePairs()* we get set O of pairs (*signature*, *name*), i.e., signature and name of referencing attributes. We get the mapping of the referenced kind $r.name_{\kappa'}$ and similarly set R of pairs of signatures and names of referenced attributes. Function *makeReferencingPairs()* processes sets O and R and creates set S of pairs *(referencing-name, referenced-name)* which is added to wrapper W_D. Finally, using function *createICStatement()* the respective command of type `ALTER KIND` is created.

Algorithm 7: IC Algorithm

Input: $m \in \mathfrak{M}$ – particular mapping
1 W_D – IC wrapper for a database D
// processing of identifier:
2 $N :=$ collectNames($m.P_\kappa$, $m.pkey_\kappa$)
3 W_D.appendIdentifier($m.name_\kappa$, N)
// processing of references:
4 **foreach** r in $m.ref_\kappa$ **do**
5 $O :=$ collectSigNamePairs($m.P_\kappa$, $r.R_{\kappa'}$)
6 $n := \mathfrak{M}$.get($r.name_{\kappa'}$)
7 $R :=$ collectSigNamePairs($n.P_\kappa$, $r.R_{\kappa'}$)
8 $S :=$ makeReferencingPairs(O, R)
9 W_D.appendReference($m.name_\kappa$, $n.name_\kappa$, S)
10 **return** W_D.createICStatement()

Multi-model-to-multi-model migration

Having both directions of transformation, i.e., to and from the categorical representation, we can now easily perform the migration between any combination of models. Instead of mutually mapping n models, i.e., to create $O(n^2)$ mappings, we only need to

map each model to the categorical representation, i.e., to create $O(n)$ mappings. This idea is not new; however, the categorical representation is sufficiently general that it covers all currently popular models (and probably many, if not all, coming in the future) and in particular their mutual combinations, i.e., inter-model references. Hence, we do not consider only model-to-model migration but more general multi-model-to-multi-model migrations. The level of abstraction enables us to "hide" many system-specific features, such as, e.g., different types of complex structures (e.g., arrays, maps, or lists), different types of links (e.g., foreign keys, references, or pointers), etc. At the same time, the abstract representation bears information that is not supported by particular underlying systems (e.g., the schema of schema-less systems or integrity constraints for inter-model links).

In the middle of Fig. 12, we can see a part of the schema category of the sample data. The colors represent mappings between the categorical representation and particular kinds (green for kind *Order* in the document model, blue for kind *Customer* in the graph model, yellow for kind *Orders* in the graph model, violet for kind *Order* in the graph model, and red for kind *Items* in the column model). For each model, we can see both the access path and the respectively highlighted part of the schema category.

For example, we may want to perform migration from the document model to a combination of the other four models. In the figure on the left, we can see the sample source (green) JSON document stored in the document model. On the bottom right we can see the target (red) column family; on the right up we can see (blue, yellow, and violet) graph data.

The migration process works as follows: Having defined all access paths, we first run the model-to-categorical transformation (see "Model-to-category transformation" section) whose result is provided in Fig. 8, i.e., we get an instance category filled with data from the underlying document DBMS (*MongoDB*). Next we run the categorical-to-model transformation (see "Category-to-model transformation" section). First, it creates the respective schemas (see "DDL algorithm" section). In the case of the schema-less graph model of *neo4j*, it does not define the structure, in the case of the column model of *Cassandra* it defines the schema of the table. Next, it stores the data instances in the DBMSs (see "DML algorithm" section) and in the last step it adds the respective integrity constraints (see "IC algorithm" section), namely command `ALTER TABLE` for *Cassandra* and no commands for *neo4j*.

All these steps were performed automatically, only with the mapping between the categorical representation and the particular DBMSs based on the idea of access paths. This is the only manual work required from the user. In addition, in the following section we introduce a user-friendly tool that enables us to specify them comfortably.

Framework *MM-cat*

As we have already mentioned, while the categorical representation and the respective mapping can be expressed manually, we do not assume that the user would do so. In this section, we show how the process can be made user-friendly using an appropriate tool.

To demonstrate the applicability of the proposed approach, we have implemented an extensible framework called *MM-cat* [10]. Its primary purpose is user-friendly modeling

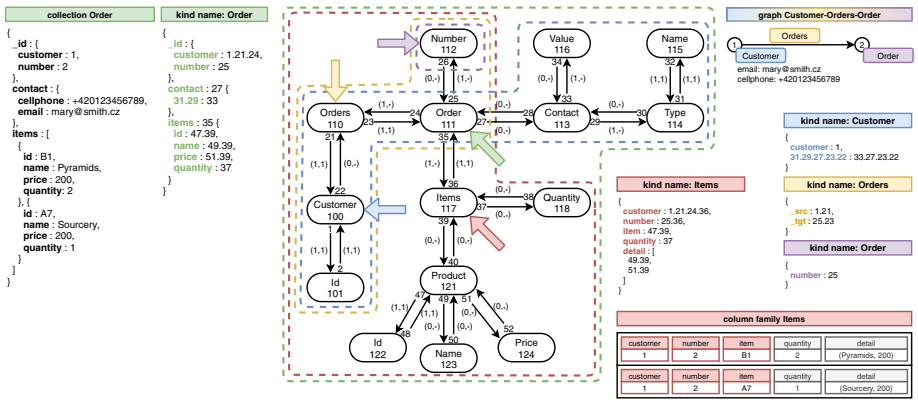


Fig. 12 Example of transformation from document model to graph and column models

of a multi-model schema and its mapping to a respective polystore, multi-model database, or a set of databases. Using the proposed transformation algorithms the user can then transform the data to/from the categorical representation. At the same time *MM-cat* serves as a basis for further possible extensions and application of the core idea in advanced data processing tasks forming our current and near-future work as discussed in Section .

The basic work with *MM-cat* assumes that the user creates a new schema from scratch. The following steps are expected to be carried out:

1 An ER schema of the target problem domain is created using usual approaches and recommendations.
2 The input ER schema is automatically transformed to schema category **S** using the algorithm proposed in [7].[14]
3 **S** is manually mapped to a selected combination of models. In particular, for each kind κ the following steps are performed:

   (a) A particular DBMS and if needed[15] a particular model is specified. Either it is already know to *MM-cat* or the user specifies the respective parameters (i.e., a connect string).
   (b) A root object of kind κ in **S** is selected and its name is specified.
   (c) The structure of κ is defined, i.e., its levels and respective properties are specified. In particular, for each property its context, name, and value is specified. The continuously evolving commands of type `CREATE KIND` and `ALTER KIND` are visualized to the user to check the correctness of the mapping.

4 The scripts with resulting commands of type `CREATE KIND` and `ALTER KIND` are generated. They can be also sent to the respective DBMS(s) to be executed. Then,

[14] An advanced user can create **S** directly, without the need to create the ER schema. We add this auxiliary step for easier understanding through a well-known approach.
[15] The combined models can be a part of separate DBMSs or a single multi-model DBMS.

the database structures (i.e., tables, collections etc.) in particular DBMS(s) as well as instance category **I** in *MM-cat* are empty.

5 The user stores data to the created database structures (using an external tool).

6 The content of the instance category **I** is created, e.g., imported from a CSV file or a particular DBMS filled with respective data.

As depicted in Fig. 13, *MM-cat* enables us to visualize and modify the current status of the multi-model modeling process. We can interactively work with the graphical representation of the ER model as well as the respective schema category. We can choose the level of detail we want to see, i.e., the amount of information provided. We can also see the JSON-like expression of the access paths and the resulting commands of type `CREATE KIND`.

For a demonstration of the key contributions of the proposed categorical approach, *MM-cat* supports two DBMSs selected to cover most of the distinct features related to multi-model data modeling—*MongoDB*[16] and *PostgreSQL*[17]. The versatility of the approach can be demonstrated from different viewpoints:

1 *Schema-less (MongoDB)*[18] *vs. schema-full/schema-mixed (PostgreSQL)*: *MM-cat* supports different approaches to the propagation of information about the specified structures to the particular DBMS. In both cases the user specifies the required structures using *MM-cat*; however, only in the case of schema-full (or schema-mixed) DBMS is the information propagated to DDL commands. In addition, dynamically derived names of properties are also not allowed, e.g., in a schema-full relational DBMS. Besides, *MM-cat* supports two cases of a schema-mixed approach:

   (a) *With a modeled schema*: The user specifies the schema, even if it is not fully propagated to the DBMS. This happens when the features of a particular DBMS do not support *schema-on-write* approach for some models (in *PostgreSQL*, it is represented by schema-less data type *JSONB* for JSON documents which can be used in a schema-full relational table). But the whole schema remains defined in the categorical representation. It can be used, e.g., for external checking of data validity of the schema-less parts, conceptual cross-model querying, etc. (This approach can also be used for a schema-less DBMS, where we want to specify the schema externally.)

   (b) *Without a modeled schema*: During the modeling phase, the user decides to leave a part of the schema unspecified, i.e., only a general data type (e.g., a *BLOB*) is assigned to a (part of a) kind. When the data stored in the DBMS is transformed to an instance category, the missing part of the schema can be inferred from the data instances. In other words, the *schema-on-read* approach is used for further processing of the data now with a known structure.

[16] https://www.mongodb.com/.

[17] https://www.postgresql.org/.

[18] For the demonstration we consider *MongoDB* as schema-less, i.e., we do not exploit its ability to define a JSON schema.

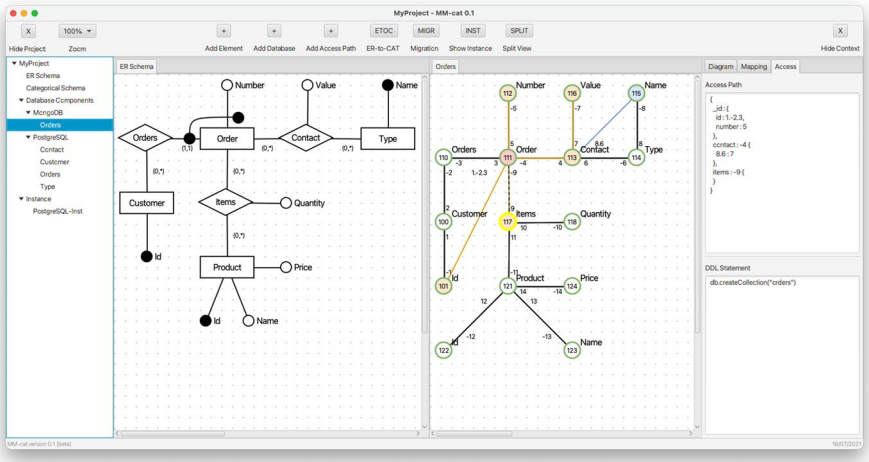


Fig. 13 Sample screen shot of *MM-cat*

2 *Aggregate-oriented (MongoDB) vs. aggregate-ignorant (PostgreSQL)*: *MM-cat* supports differences in the mapping process regarding the complexity of structures allowed by the particular type of a system. Both complex hierarchical structures allowing nesting and repetitions (arrays) and flat relations with only simple data types (or their combination in the case of multi-model *PostgreSQL*) can be created.
3 *Polystore vs. multi-model DBMS*: *MM-cat* can handle modeling of a schema in the case of a polystore-like approach, i.e., combining models from several DBMSs, and in the case of a single multi-model DBMS which is capable of storing multiple models in a single system.

Architecture and implementation

MM-cat was implemented using *Java SE 16*, graphical library *JavaFX*[19], and *Apache Maven*[20]. For communication with *MongoDB* and *PostgreSQL* we use the respective Java (JDBC) drivers[21,22].

The architecture of the framework is depicted in Fig. 14. At the bottom we can see *n* (green) DBMSs which represent all possible combinations of usage of multiple models, i.e.:

1 a multi-model DBMS,
2 a set of single-model DBMSs, or
3 a combination of the previous two cases.

[19] https://openjfx.io/.
[20] https://maven.apache.org/.
[21] https://mongodb.github.io/mongo-java-driver/.
[22] https://jdbc.postgresql.org.

For a unified access, each of the DBMSs is wrapped using a unified interface providing functions for defining a schema and integrity constraints, defining mapping to categorical representations, and storing/extracting data. Each system-specific (green) wrapper implements an interface of the respective abstract (grey) wrapper. The yellow boxes represent the core categorical data structures defined in "Categorical representation of multi-model data" and "Category-to-data mapping" sections and , i.e. the schema category, the instance category, and the access paths representing the core of the mapping. The transformation between the categorical structures and the wrappers representing the DBMSs (described in "Transformation" section) is ensured by the two blue transformation modules.

Finally, we also depict the red modules which represent the advanced functionality that we are currently implementing on top of the categorical data structures and transformation modules, i.e.

1. *conceptual querying* over the categorical representation,
2. *inference of a categorical schema* from data instances,
3. *migration of data* between different DBMSs (having the same or distinct model), and
4. *evolution management*, i.e., propagation of user-specified changes in the categorical schema to affected parts (i.e., primarily data instances and operations).

The unified representation of the data enables us to work with any combination of the underlying models regardless of implementation-specific details of particular systems.

Wrappers

A wrapper represents a bridge between a particular DBMS and the unified categorical layer. Each wrapper implements a selected interface of an abstract wrapper, namely:

1. `AbstractPullWrapper` for extracting data from a DBMS (i.e., calling queries of type `SELECT * FROM KIND ...`),
2. `AbstractPushWrapper` for storing data into a DBMS (i.e., calling commands of type `INSERT VALUES (...) INTO KIND ...`),
3. `AbstractICWrapper` for adding integrity constraints (i.e., calling commands of type `ALTER KIND ... ADD CONSTRAINT ...`),
4. `AbstractDDLWrapper` for the definition of a schema (i.e., calling DDL commands of type `CREATE KIND ...` without integrity constraints), and
5. `AbstractPathWrapper` for the definition of mapping to categorical structures which differ in the particular DBMSs (models), e.g., in the (dis)allowed nesting of properties.

On top of the wrappers, we primarily implement the proposed transformation algorithms (but other functionalities can be implemented on top of them too) in a unified way, i.e., regardless the specifics of the underlying DBMS. Moreover, adding new DBMS does not require changes in the higher-level modules, only the new wrappers need to be implemented. The underlying system does not need to be a particular existing DBMS, but it can be, e.g., a file manager ensuring the functionality of the unified interface.

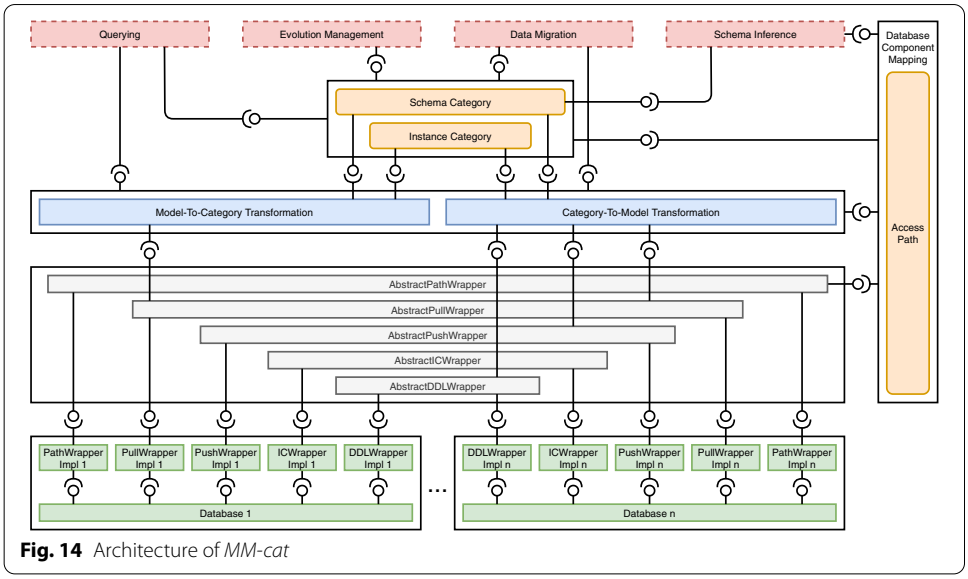


Fig. 14 Architecture of *MM-cat*

`AbstractPathWrapper` From the point of view of the proposed categorical representation, the most interesting wrapper is `AbstractPathWrapper`. As we can see in Table 2, it returns information about the allowed complexity of the mapping in the particular DBMS.

For example, in the case of *MongoDB* its `MongoDBPathWrapper`[23] enables to inline properties without any restrictions. When a morphism with the upper bound of a cardinality > 1 occurs on the path to the inlined property, an array of the inlined properties is created. The wrapper also enables the grouping of selected properties into an auxiliary property not defined in schema category **S**.

For *PostgreSQL*, its `PostgreSQLPathWrapper`[23] enables inlining only when the upper bounds of morphisms have cardinality $= 1$ (since arrays are not allowed). It also does not allow grouping or complex nested structures (since relational tables are flat). Dynamically derived names and anonymous names are not allowed too, due to the features of the relational model.

The abstract wrapper also predefines the following methods:

- Method `addProperty(String hierarchy)` adds a new property to the currently constructed access path. Parameter `hierarchy` contains its hierarchical name (e.g., `/Order/Items/_/Name`).
- Method `check()` enables to check whether the currently constructed access path follows requirements of the particular DBMS. For example, in the case of *MongoDB* it checks whether compulsory property *_id* being the identifier is present.

`AbstractDDLWrapper` The methods that are used in Algorithm 4 (DDL Algorithm) are predefined by the `AbstractDDLWrapper`. In particular they involve the following ones:

[23] Which implements `AbstractPathWrapper`.

Table 2 Allowed complexity of mapping in *MongoDB* and *PostgreSQL*

	MongoDB	***PostgreSQL***
`isRootObjectAllowed()`	True	True
`isRootMorphismAllowed()`	True	True
`isPropertyToOneAllowed()`	True	True
`isPropertyToManyAllowed()`	True	False
`isInliningToOneAllowed()`	True	True
`isInliningToManyAllowed()`	True	False
`isGrouppingAllowed()`	True	False
`isDynamicNamingAllowed()`	True	False
`isAnonymousNamingAllowed()`	True	False
`isReferenceAllowed()`	True	True

- Method `setKindName(String name)` denotes the name of a kind (i.e., table, collection, etc.) for which the schema is created.
- Method `isSchemaLess()` determines whether the creation of a schema is (not) required, i.e., the database implements a schema-less or a schema-full approach.
- Method `addSimpleProperty(Set<String> names, boolean optional) throws UnsupportedOperationException` enables the creation of a property with a simple data type. Usually a separate property is created for each value in parameter `names`. But, for example, in the case of *Cassandra* the wrapper-specific behaviour ensures that a multi-value property is transformed to a map which influences the parent property as well. Parameter `optional` denotes whether value `null` is allowed.
- Method `addSimpleArrayProperty(Set<String> names, boolean optio-nal) throws UnsupportedOperationException` creates an array of simple data types.[24]
- Method `addComplexProperty(Set<String> names, boolean optional) throws UnsupportedOperationException` creates a property with a complex type (structure).
- Method `addComplexArrayProperty(Set<String> names, boolean optio-nal) throws NotAllowedException` creates a property with an array of complex types. (Note that we distinguish an array of simple types and an array of complex types, because in some systems, e.g., *neo4j*, only the former one is allowed.)
- Method `createDDLStatement()` creates and returns the resulting DDL command for a particular DBMS.

For example, `MongoDBDDLWrapper`[25] implements only method `setKind-Name()`, whereas the remaining ones are empty (because *MongoDB* is schema-less) and do not throw any exception (because *MongoDB* is aggregate-oriented). Method

[24] Parameters `names` and `optional` have the same behaviour as in the previous case.

[25] Which implements `AbstractDDLWrapper`.

`createDDLStatement()` then returns only command `createCollection` with the respective name of the collection.

`PostgreSQLDDLWrapper` implemented purely for the relational model in *PostgreSQL* implements methods `setKindName()`, `addSimpleProperty()` (adding a simple property with kardinality (1,1)), and `addSimpleArrayProperty()` (*PostgreSQL* supports arrays of simple types). However, in case of the other methods the wrapper throws an exception `UnsupportedOperationException`, since it is aggregate-ignorant. (Note that the wrapper, e.g., for *neo4j* would have similar behaviour.) Method `createDDLStatement()` returns the respective command `CREATE TABLE` without integrity constraints.

`AbstractPushWrapper` Methods used in Algorithm 4 (DML Algorithm) are predefined by the `AbstractPushWrapper`. In particular they involve the following ones:

- Method `setKindName(String name)` denotes the name of the kind (i.e., table, collection, etc.) where the instances are stored.
- Method `append(String name, Object value)` appends `value` associated with the `name` to the currently created DML command.
- Method `createDMLStatement()` returns the resulting DML command.
- Method `clear()` removes all the data previously added to create a DML command, i.e., the name of the kind and (*name*, *value*) pairs from the currently created DML command.

Wrappers for all types of DBMS implement methods `setKindName()`, `append()`, and `removePairs()` in the same way. Naturally the key difference is in method `createDMLStatement()` which is strongly system-dependent. For example, *PostgreSQL* wrapper transforms pairs (`name_1`, `value_1`), ..., (`name_n`, `value_n`) of kind κ to command `INSERT INTO KIND` κ `(name_1, ..., name_n) VALUES (value_1, ..., value_n)`. On the other hand, *MongoDB* wrapper creates command `db.collections.insert(...)`, where `name_i`, `i` $= 1, ...n$ denote names of fields in the hierarchy and `value_i` denote their respective values.

`AbstractICWrapper` Methods used in Algorithm 7 (IC Algorithm) are predefined by the `AbstractICWrapper`. In particular they involve the following ones:

- Method `appendIdentifier(String name, IdentifierStructure pid)` enables the addition of an integrity constraint representing an identifier to kind specified using parameter `name`. The structure of the identifier is provided in parameter `pid`. The structure defines not only the set of properties forming the identifier, but also their order and nesting, depending on the requirements of the particular DBMS.
- Method `appendReference(String name, String name2, Set<Pair<String, String>> atts)` enables adding of a reference from referencing kind specified using parameter `name` to referenced kind specified using parameter `name2`. Parameter `atts` contains a set of pairs *(name of referencing property, name of referenced property)*.
- Method `createICStatement()` creates a set of commands of type `ALTER KIND` for adding specified integrity constraints.

- Method `createICRemoveStatement()` creates a set of commands of type `ALTER KIND` for (temporary) removal of specified integrity constraints.

The system-dependent processing of integrity constraints strongly differs. For example in *MongoDB* there is a compulsory property *_id* which is checked by `MongoDBPathWrapper` in function *check()*. In *PostgreSQL* the selected properties forming the primary key or the foreign key are denoted in the command `ALTER TABLE`. In schemaless *MongoDB* the references do not modify the schema at all.

`AbstractPullWrapper` Last but not least, `AbstractPullWrapper` predefines the methods used in Algorithm 1 (Model-to-Category Transformation) for the construction of the forest of records. In particular:

- Method `pullForest(String selectAll, AccessPath path)` first extracts all records using a database-specific command `selectAll`. Then, using the information from the access path `path`, it transforms each of the records to a respective tree and adds it to the resulting forest.
- Method `pullForest(String selectAll, AccessPath path, int limit, int offset)` has the same behavior. In addition, it enables the ability to set `limit` and `offset` for pagination in case the particular system supports this feature.

Performance

The algorithm's complexity depends on whether we need to index the identifiers of the records of particular kinds. If not, it is linear regarding the number of records in the input data set, i.e., all kinds. If so, the respective records need to be indexed for each kind. In other words, for each kind κ with N_κ records we get $O(N_\kappa.log(N_\kappa))$ instead of $O(N_\kappa)$ without an index. None of the steps require complex modifications of the existing instance category in both cases. In addition, the algorithms are designed to be simply transformed into a parallel version and thus scalable.

- Model-to-category transformation (Algorithm 1) can be parallelized to process very large collections of data or very large data files.
  - A large collection of input records to be transformed can be split into subsets processed by multiple threads, each applying the algorithm on a particular subset individually. The only requirement is to avoid conflicts in method *modifyActiveDomain()*, where, e.g., *Java Atomic Classes*[26] and a lock-free approach can be used.
  - A single large document can also be split to be processed in parallel by multiple threads exploiting the stack M and a parallelized DFS algorithm as every subtree of the access path is independent of its sibling subtrees.

[26] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/atomic/package-summary.html.

- DDL algorithm 4 only defines and creates a schema of the data (i.e., its structure), therefore a scalable implementation is not considered due to the nature of the schema, i.e., a small set of possibly nested simple or complex properties comparable in size to a single record.
- DML algorithm 5 is parallelizable depending on whether $morph_\kappa$ is *null* or not. If it is *null*, then line 13 can be parallelized, i.e., the active domain of q_I can be distributed across multiple threads to be processed by the foreach cycle. Otherwise, similarly, relations from line 19 can be distributed across multiple threads to be processed in parallel at line 21.
- IC algorithm 7 only generates statements of type `ALTER KIND`, therefore it is not considered being parallelized.

The still gradually improved implementation of the approach, *MM-cat*, contains various technical tricks enabling further optimization. For example, in the case of entries in the active domain of objects, we assume an optimistic approach and, therefore, a lock-free approach (i.e., no synchronization, no locks), implemented using *Java Atomic Classes*. The probability that we will work with the same memory space simultaneously is minimal, so there is no need to synchronize larger sections of code (i.e., to lock them). This can only happen in the case of method *modifyActiveDomain()*, where we can merge existing mappings (rows of active domains).

Or, we assume that only the active domains of some objects need to be indexed—e.g., identifiers of complex structures or attributes influencing querying efficiency.

Or, composite morphisms do not have to be explicitly materialized into a mapping. We use a lazy strategy, where only a repeatedly used composite morphism is materialized.

Benefits of category theory

To conclude the description of the proposed approach, we discuss the main benefits of the utilization of category theory. At first sight, it may seem that the existing models are rich enough to be used as a mediator for the representation of multi-model data. Unfortunately and naturally, none of the popular models covers all specifics of the others and such a transformation would lead to complex or unnatural and thus inefficient constructs. For example, we can consider the well-known issue of representing graph data in the relational model or the large difference between aggregate-oriented and aggregate-ignorant models and the respective (de)normalization of data. More abstract data representations also exist, however, their expressive power is still limited as we discuss in [11].

The next important aspect is the further exploitation of the categorical representation. Our aim is not only to find a "tool" for representing multiple interconnected models. As we have discussed in [12], this unified representation enables one to perform further data management tasks, such as cross-model querying or evolution management, uniformly, correctly, and efficiently. Although these extensions form our (near) future fork, in the following section we provide an example that demonstrates the indicated advantages, namely, in the case of querying.

Application—querying

To demonstrate how the categorical querying over the proposed categorical framework could work, let us again consider the following sample multi-model query [1]: "For each customer who lives in Prague, find a friend who ordered the most expensive product among all customer's friends." As for the result and its representation, Fig. 15 depicts a possible schema of the projected properties including its mapping to the output graph model representation (hence depicted in the blue color). In addition, the figure illustrates the multiple logical models incorporated in the query:

- The relational model represents the data about customers and their addresses, i.e., kinds *Customer* and *Address*. (Note that the exploitation of a composite morphism enables us to directly "access" object *City*.)
- The graph model represents the data about customers and their friends, i.e., kinds *Customer* and *Friend*.
- The column model represents the relationships between customers and their orders, i.e., kinds *Customer* and *Order*.
- The document model represents the order that (possibly) consists of multiple ordered products, i.e., kinds *Order* and *Product*.

Note that for clarity and simplicity, the objects of the projection schema are labeled with the same signatures as the corresponding objects of the schema category. The apostrophe "′" is added to distinguish unique labels in the case of duplicity caused by morphism *friend* with the same source and target object *Customer*. The same applies to morphisms.

Finally, during the evaluation of the query, one may exploit the fact that the identifier of kind *Customer* is a part of the identifier of kind *Order*, therefore the query evaluation does not have to consider data from the column model. In other words, there is an opportunity to exploit overlapping data for different evaluation strategies.

Query execution

The execution of the sample query would consist of multiple stages:

I. Query Pattern and its Mapping First, a pattern describing the query is created. As proposed in our previous work [12], a query pattern could be represented in the form of a *query category* structurally corresponding to a part of the schema category. In other words, there is a functor between the query category and schema category.

In general, the query pattern could be similar to the projection schema from Fig. 15, but additionally enriched by query operators (e.g., union, aggregation, or filtering condition), all represented in the form of additional categorical objects or morphisms.

The idea of a categorical query language is not new [13]. In comparison to, e.g., *Cypher* [14] the advantage of categorical representation is the possibility to exploit composite morphisms which simplify the structure of a query. Hence, a complex graph traversal can be represented by a single composed morphism—e.g., $39.35.23.22 : \mathit{Customer} \rightarrow \mathit{Product}$ can be used to express the traversal from *Customer* to *Product*, corresponding to the composition of morphisms `22`, `23`, `35`, and `39`. (Note

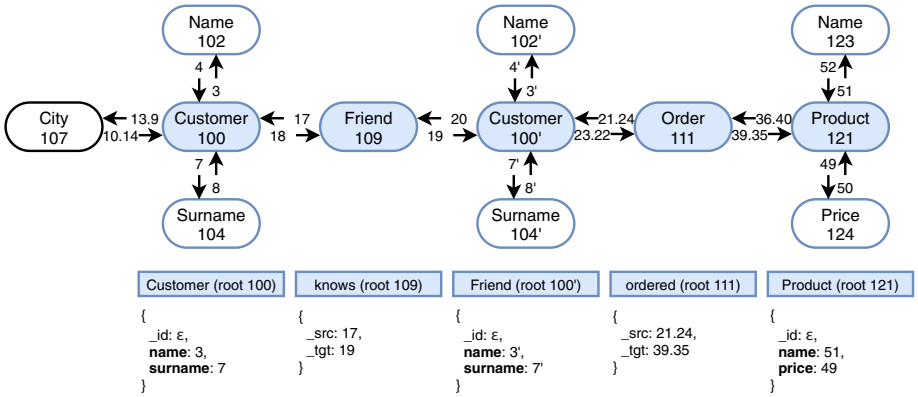


Fig. 15 Projection of the result corresponding to the graph pattern Customer—knows→Friend—ordered→Product and its mapping to the graph model

that we use two morphisms in the example, namely, 23.22 : *Customer* → *Order* and 39.35 : *Order* → *Product* to represent the same path.)

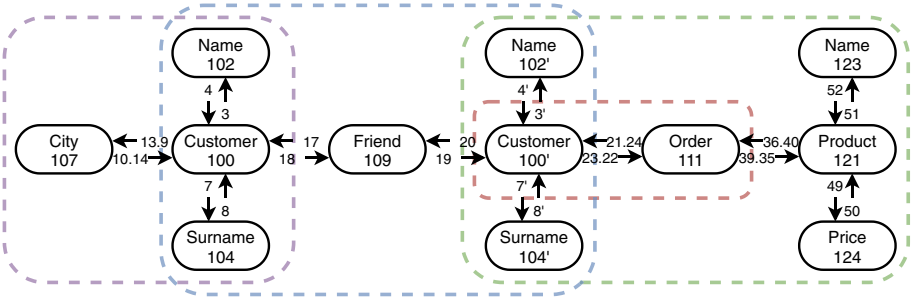
II. Query Decomposition The decomposition of a query into so-called *query parts* [12] exploits the functor between the query pattern and the schema category and the mapping of the schema category to particular databases (or their specific models) to determine which query parts will be executed under which logical data model. As denoted in Fig. 16, a possible decomposition of the sample query could be done as follows:

- Customers living in Prague will be evaluated in the relational model, i.e., the query part will be translated into an SQL statement.
- Friends of the customers will be evaluated in the graph model, i.e., the query part will be translated, e.g., into *Cypher* statement.
- The most expensive product ordered by a customer will be evaluated in the document model, i.e., an aggregation query will be translated into, e.g., the *MongoDB* query language [15].

Note that the column database in our sample scenario can be exploited in an alternative query plan to simplify the query evaluation in the document database to match a particular customer with all his/her orders. Moreover, note that the projection of attributes in each query part forms a subset of objects of the query pattern category.

III. Evaluation of Query Parts If independent of each other, the evaluation of query parts can be executed in parallel and the partial results are then joined and merged. During the execution of each query part (translated into a particular query language or at least constructs specific for the corresponding logical data model), we can utilize existing approaches and exploit all benefits of its logical representation, including single-model query execution plans and management.

IV. Unification and Joining of Intermediate Results Each query part produces a result translated to appropriate objects and morphisms in the query pattern using the model-to-category transformation. The main benefit of unifying categorical representation is the simple joining of partial results. Considering other models, having all partial results represented in the relational model, its joining could be expensive, e.g., without having




```
SELECT customerId
FROM Customer
WHERE city = "Prague";
```

```
MATCH
  (c:CUSTOMER)
    -[:KNOWS]->
      (f:CUSTOMER)
RETURN
  c.id, c.name, c.surname,
  f.name, f.surname, f.id;
```

```
db.orders.aggregate( [
  { $unwind : "$items" },
  { $sort : { "items.price" : -1 } },
  { $group : {
    _id : "$_id.customerId",
    items : { $push : { name : "$items.name", price : "$items.price" }}
    } },
  { $project: {
    _id : 1,
    name : { $arrayElemAt : [ "$items.name", 0 ] },
    price : { $arrayElemAt : [ "$items.price", 0 ] } }
} ] );
```

Fig. 16 The decomposition of projection pattern and examples of translations of corresponding query parts into particular statements

respective indices. Similarly, the joining of aggregates can also be expensive due to possibly denormalized and redundant data. Or, joining in the graph model may require a special edge that connects two objects that are otherwise not connected.

On the contrary, the unifying categorical approach allows us to join the corresponding parts of the intermediate results of cross-model queries easily using so-called *pullbacks* [16, 17], i.e., a generalization of the Cartesian square and intersection. As illustrated in Fig. 17 using the respective colors, there will be two pullbacks to join partial results between the relational and graph model (i.e., $P_1 = result_{REL} \bowtie_{100} result_{GRAPH}$) and between the result of the first pullback and document model (i.e., $P_2 = P_1 \bowtie_{100'} result_{DOC}$).

In general, the joining of the intermediate results may be processed in an arbitrary order. However, the selected strategies and joining execution plan should be considered to reduce the time complexity. Nevertheless, multi-model joins add a new level of complexity to querying [18] and form a largely open research area.

V. Transformation to the Desired Representation Finally, we transform the categorical representation to the requested logical model representation. In the sample query, the result is transformed into a graph representation as illustrated in Fig. 18.

Alternative Multi-Model Query Plan Alternatively, since we have overlapping data in the document and column model, we could use a different query evaluation strategy. It would simplify the aggregate query in the document model, but at the cost of one more join of results from different data models, i.e., an additional pullback and possibly a large amount of data to be joined at the level of the unifying model.

In general, the strong point of categorical querying over the categorical representation is that the user does not explicitly have to know the logical representation of the data. E.g., having a query over multi-model *PostgreSQL*, the user still has to be aware of the data logical representation, thus (s)he must decompose the query into the relational,

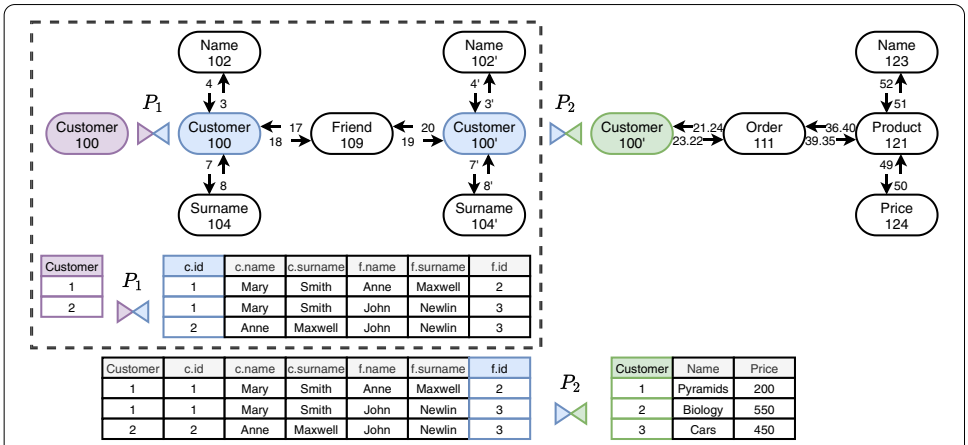


Fig. 17 Joining of intermediate results by pullbacks

JSON, and XML parts and use model-specific query constructs for them. Categorical representation allows one to use unified query constructs across all models, then internally translated to model-specific constructs.

The graph representation of the categories is natural and enables one to cover all popular data models. In addition to the graph model, the categorical representation involves several extensions, such as complex or overlapping identifiers, required in other models. And what is most important, the theory behind enables us to process the data easily, e.g., using composite morphisms.

Related work

Each of the existing multi-model DBMSs [3] naturally (and more or less painfully) provides an extension of the original data structures used for a single core model. There also exist proposals of more general approaches. E.g., the *NoSQL Abstract Model* [19] represents the data as named collections, each containing a set of blocks consisting of a non-empty set of entries. *Associative arrays* [20] are defined as mappings from pairs of unique (column and row) keys to values. Or, the *Tensor Data Model* [21] introduces the idea of generalized matrices. However, we need to target a more abstract level for a truly universal approach covering the specifics of various common data models and especially their combinations.

In the context of polystores, *TyphonML* [22] enables us to specify conceptual entities, their attributes, relations, and datatypes, and map them to different single-model DBMSs of a polystore. Similarly, in paper [23], an ER schema is partitioned and then mapped to different data models. However, none of these approaches provides a detailed specification of how the respective inter-model references should be managed, whether overlapping is supported, how cross-model querying will be handled, etc. There also exist older proposals which, however, consider only earlier database systems and respective models [24–26]. Recently, paper [27] introduced the notion of *U-Schema*, involving entity type, simple and multivalued attributes, key attribute, and three kinds of relationships between entity types: aggregation, reference, and inheritance. In addition, there are relationship types and structural variations of entity and relationship types. The authors show the mapping between U-Schemas and common data models in both directions.

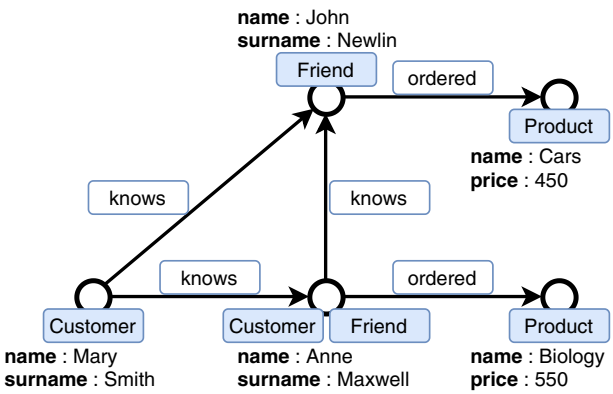


Fig. 18 The result of the query represented in a graph model

However, in this case, the consideration of inter-model links and related aspects is limited. In addition, despite the authors trying to ensure unification of the models, they involve special constructs that cover specific features of particular models. On the contrary, we provide a general abstract representation of popular models based on the natural notion of a graph that covers all the indicated issues.

The idea of exploiting category theory to represent data models is not new. Most of the approaches, denoted as *bottom-up*, start from a single logical model (namely, relational [16, 28], or object-relational, i.e., hierarchies of classes [29]) and define a respective schema category and operations using standard categorical approaches (such as functors). Paper [30] proposes a categorical approach for relational (CSV), document, and graph (RDF) models, but only with intra-model data migrations and querying. A *top-down* approach from [31] defines a schema category covering various conceptual modeling approaches, but unfortunately only concerning the most common model of that time—relational. The exploitation of category theory for multi-model data is so far quite limited. On the contrary, in our proposal we cover all the currently popular models together with respective inter-model links, i.e. a truly multi-model solution.

Conclusion

In this paper, we continue to build a general framework for unified modeling and management of multi-model data. We believe that category theory is the right "tool" for representing various data models using a rigorously defined and sufficiently general theory. This text shows that it enables us to grasp and process the varying nonstandardized multi-model world uniformly and precisely.

The proposed approach, implemented in *MM-cat*, has several important advantages for multi-model data modeling:

1 It enables us to model the multi-model schema using a data structure which:

   (a) can be automatically extracted from a well-known conceptual model (e.g., ER),
   (b) is enough general to cover all known models, and
   (c) is based on a well-known notion of a graph.

2 It enables us to map the conceptual model of the data to any (combination of) DBMSs and respective models, whereas the user does not need to deal with implementation specifics.
3 Besides the schema category which describes the schema of the data, the instance category serves as a mediator which enables the unified representation of an instance of the data. It is expected to be materialized only to the necessary extent, e.g., to represent intermediate results of queries.

The core categorical approach also provides a range of applications simplifying and optimizing various aspects of multi-model data management:

- *Conceptual Query Language:* The level of abstraction of the proposed categorical approach enables one to define a *conceptual query language* that can be mapped to any multi-model query language. In addition, since a graph backs the categorical model, the query language might be inspired by graph query languages like, e.g., *Cypher* [14] or *SPARQL* [32] and thus naturally adopted by the users. The conceptual queries can be translated to expressions required by a particular DBMS using a similar strategy.
- *Data Migration:* Migration of data between various DBMSs (with the same or distinct data models) can be done much easier with the unified categorical representation of any (combination of) data models. The user only specifies another mapping between the schema category and the target model.
- *Evolution Management:* Having the unified categorical representation, both intra and inter model modifications of the schema are reduced to the same task—modification of a graph representing the schema category and a respective propagation of changes to all affected parts. Again, the key issues are related to the mapping between the categorical and logical representation which needs to be extended by the user when needed.
- *Extensibility*: Since the categorical model is defined universally for any data model, it can then be applied to any multi-model DBMSs. We do not define special constructs for particular models (such as, e.g., the relationship type in [27]). In addition, the idea enables one to cover even data models that are not currently known; the only requirement is that they can be described using the same categorical structures.

Finally, note that the approach is also applicable for single-model systems. Thanks to the unification of the models, it can be applied to both NoSQL databases and traditional relational databases. A single-model system can be managed separately; however, a more probable approach can reflect the idea of polyglot persistence, where multi-model data is stored in several single-model systems, each suitable for a particular part of the data.

Future work

As indicated before, in the (current and) future work, we will primarily aim at correct and efficient evolution management and data migration. Our second target is a conceptual query language that would enable us to query across the distinct data models

without knowing their specifics. In both cases, we can directly exploit the features of the proposed framework.

On the other hand, even the core idea can be further extended. Some of the extensions may involve:

- *Simple types*: In the current proposal, we consider a basic set of simple types, i.e., string and numeric. However, the existing DBMSs support various simple types, even with distinct features.
- *Cardinalities*: The set of supported cardinalities can be extended with other types, such as, e.g., numeric specification of the bounds, a set of bounds, etc. The respective composition of morphisms then needs to be extended and special cases for particular models must be reflected in the wrappers.
- *Aliasing*: The dynamically derived names could have also cardinality (1, *N*) and thus enable a kind of *aliases*, i.e., naming the same data differently for different purposes, e.g. simpler expression of queries.
- *Evaluation of values*: The access paths could be extended with both constant values and basic functions for the evaluation of new values (e.g., various aggregations).

Abbreviations
DBMS: Database management system; ER: Entity-relationship; UML: Unified modeling language; RDF: Resource description framework; ISA: "is a"; JSON: Javascript object notation; DFS: Depth-first search; DDL: Data definition language; BLOB: Binary large object; IC: Integrity constraint.

Acknowledgements
We would like to thank Martin Svoboda for consultation and improvement of basic definitions.

Author Contributions
The share of the author's contributions in this paper is equal. All authors read and approved the final manuscript.

Funding
The work was supported by the GAČR project no. 20-22276S.

Availability of data and materials
Framework ***MM-cat*** was described in demo paper [10].

Declarations

Ethics approval and consent to participate
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Consent for publication
Not applicable.

Received: 19 September 2021 Accepted: 11 April 2022
Published online: 03 May 2022

References

1. Zhang C, Lu J, Xu P, Chen Y. UniBench: a benchmark for multi-model database management systems. In: Technology conference on performance evaluation and benchmarking '18. Lecture notes in computer science, vol. 11135. Cham: Springer. 2018. p. 7–23. https://doi.org/10.1007/978-3-030-11404-6_2

2. Kolev B, Pau R, Levchenko O, Valduriez P, Jiménez-Peris R, Pereira JO. Benchmarking polystores: the CloudMdSQL experience. In: 2016 IEEE international conference on big data (big data). New York, NY: IEEE; 2016. p. 2574–2579. https://doi.org/10.1109/BigData.2016.7840899
3. Lu J, Holubová I. Multi-model databases: a new journey to handle the variety of data. ACM Comput Surv. 2019. https://doi.org/10.1145/3323214.
4. Lu J, Liu ZH, Xu P, Zhang C. UDBMS: road to unification for multi-model data management. In: ER '18 workshops. Lecture notes in computer science, vol. 11158. Cham: Springer; 2018. p. 285–294. https://doi.org/10.1007/978-3-030-01391-2_33
5. Feinberg D, Adrian M, Heudecker N, Ronthal AM, Palanca T. Gartner magic quadrant for operational database management systems. 2015.
6. Thalheim B. Entity-relationship modeling: foundations of database technology. 1st ed. Berlin, Heidelberg: Springer; 2000.
7. Svoboda M, Čontoš P, Holubová I. Categorical modeling of multi-model data: one model to rule them all. In: model and data engineering. Lecture notes in computer science. Cham: Springer; 2021. p. 190–198. https://doi.org/10.1007/978-3-030-78428-7_15
8. Barr M, Wells C. Category theory for computing science, vol. 1. New York: Prentice Hall; 1990.
9. Hoare CAR. Notes on an approach to category theory for computer scientists. In: Constructive methods in computing science. Berlin, Heidelberg: Springer. 1989. p. 245–305. https://doi.org/10.1007/978-3-642-74884-4_9
10. Koupil P, Svoboda M, Holubova I. MM-cat: a tool for modeling and transformation of multi-model data using category theory. In: 2021 ACM/IEEE international conference on model driven engineering languages and systems companion (MODELS-C). New York, NY: IEEE; 2021. p. 635–639. https://doi.org/10.1109/MODELS-C53483.2021.00098
11. Holubova I, Contos P, Svoboda M. Multi-model data modeling and representation: state of the art and research challenges. In: 25th international database engineering & applications symposium. IDEAS 2021. New York, NY: Association for Computing Machinery; 2021. p. 242–251. https://doi.org/10.1145/3472163.3472267
12. Holubova I, Contos P, Svoboda M. Categorical management of multi-model data. In: 25th international database engineering & applications symposium. IDEAS 2021. New York, NY: Association for Computing Machinery; 2021. p. 134–140. https://doi.org/10.1145/3472163.3472166
13. Brown KS, Spivak DI, Wisnesky R. Categorical data integration for computational science. Comput Mater Sci. 2019;164:127–32. https://doi.org/10.1016/j.commatsci.2019.04.002.
14. Neo4j Inc. Cypher query language. Neo4j, Inc. 2021. https://neo4j.com/developer/cypher/
15. MongoDB, Inc. MongoDB manual—query documents. MongoDB, Inc. 2017. https://docs.mongodb.com/manual/tutorial/query-documents/
16. Spivak DI, Wisnesky R. Relational foundations for functorial data migration. In: Proceedings of the 15th symposium on database programming languages. DBPL 2015. New York, NY: Association for Computing Machinery; 2015. p. 21–28. https://doi.org/10.1145/2815072.2815075
17. Lu J, Holubová I. Multi-model data management: what's new and what's next? In: Proceeding of the 20th international conference on extended databases. 2017. p. 602–605.
18. Zhang C, Lu J. Holistic evaluation in multi-model databases benchmarking. Distrib Parallel Databases. 2019;39:1–33. https://doi.org/10.1007/s10619-019-07279-6.
19. Atzeni P, Bugiotti F, Cabibbo L, Torlone R. Data modeling in the NoSQL world. Comput Stand Interfaces. 2020;67:103–49. https://doi.org/10.1016/j.csi.2016.10.003.
20. Kepner J, Chaidez J, Gadepally V, Jansen H. Associative arrays: unified mathematics for spreadsheets, databases, matrices, and graphs. CoRR **abs/1501.05709** 2015. arxiv:1501.05709
21. Leclercq E, Savonnet M. TDM: a tensor data model for logical data independence in polystore systems. In: Heterogeneous data management, polystores, and analytics for healthcare. Cham: Springer; 2019. p. 39–56. https://doi.org/10.1007/978-3-030-14177-6_4
22. Basciani F, Di Rocco J, Di Ruscio D, Pierantonio A, Iovino L. TyphonML: a modeling environment to develop hybrid polystores. In: Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems: companion proceedings. New York, NY: Association for Computing Machinery; 2020. p. 1–5. https://doi.org/10.1145/3417990.3421999
23. Kolonko M, Müllenbach S. Polyglot persistence in conceptual modeling for information analysis. In: 2020 10th international conference on advanced computer information technologies (ACIT). New York, NY: IEEE; 2020. p. 590–594. https://doi.org/10.1109/ACIT49673.2020.9208928
24. Hick J-M, Hainaut J-L. Strategy for database application evolution: the DB-MAIN approach. In: Conceptual modeling—ER 2003. Lecture notes in computer science, vol. 2813. Berlin, Heidelberg: Springer. 2003. p. 291–306. https://doi.org/10.1007/978-3-540-39648-2_24
25. Atzeni P, Gianforme G, Cappellari P. A universal metamodel and its dictionary. In: Transactions on large-scale data- and knowledge-centered systems I, vol. 1. Berlin, Heidelberg: Springer; 2009. p. 38–62. https://doi.org/10.1007/978-3-642-03722-1_2
26. Kensche D, Quix C, Chatti MA, Jarke M. GeRoMe: a generic role based metamodel for model management. J Data Semant VIII. 2007;8:82–117. https://doi.org/10.1007/978-3-540-70664-9_4.
27. Candel CJF, Ruiz DS, García-Molina J. A unified metamodel for NoSQL and relational databases. CoRR abs/2105.06494 2021. arxiv:2105.06494
28. Schultz P, Spivak DI, Vasilakopoulou C, Wisnesky R. Algebraic databases. Theory Appl Categ. 2017;32(16–19):547–619 arxiv:1602.03501.
29. Tuijn C, Gyssens M. CGOOD, a categorical graph-oriented object data model. Theor Comput Sci. 1996;160(1):217–39. https://doi.org/10.1016/0304-3975(95)00089-5.
30. Thiry L, Zhao H, Hassenforder M. Categories for (big) data models and optimization. J Big Data. 2018;5(1):1–20. https://doi.org/10.1186/s40537-018-0132-9.
31. Lippe E, Ter Hofstede AHM. A category theory approach to conceptual data modeling. RAIRO Theor Inf Appl Inf Théor Appl. 1996;30(1):31–79.

32. World Wide Web Consortium: SPARQL query language for RDF. World wide web consortium. 2008. http://www.w3.org/TR/rdf-sparql-query/

Publisher's Note