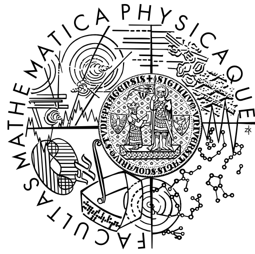


FACULTY OF MATHEMATICS AND PHYSICS
Charles University

DOCTORAL THESIS

Pavel Koupil (Čontoš)

Modelling and Management of Multi-Model Data

Department of Software Engineering

Supervisor of the doctoral thesis: doc. RNDr. Irena Holubová, Ph.D.
Study programme: Computer Science
Study branch: Software Systems

Prague 2022

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

First and foremost, I would like to thank my supervisor Irena Holubová for her invaluable advice, constructive feedback and support during my PhD studies. Despite all the complications that the pandemic brought with it, our cooperation was always smooth and friendly.

I also wish to thank all my co-authors. Martin Svoboda for his advice especially during the beginning of my studies. Ivan Veinhardt Latták, Sebastián Hricko, and Jáchym Bártík for their contributions, which took our knowledge a step further. My thanks also belong to all the anonymous reviewers for their constructive comments and suggestions during the publication of the papers.

I would also like to thank to members of L1 Allegra Laser Group from the project ELI Beamlines who allowed me to find the time to pursue my PhD studies.

Last but not least, I am deeply grateful to my wife. Without her tremendous understanding and encouragement over the past few years, it would not be possible for me to complete my studies. We tried our best to support each other in our individual PhD journeys and we shared our joys and sorrows alike.

Dedicated to my father (1958-2016) and ...

Title: Modelling and Management of Multi-Model Data

Author: Pavel Koupil (Čontoš)

Department: Department of Software Engineering

Supervisor: doc. RNDr. Irena Holubová, Ph.D., Department of Software Engineering

Abstract: With the advent of multi-model database management systems, the boundaries of many approaches to data processing were pushed. The aspect of multi-model data introduces a new dimension of complexity and new challenges not seen in single-model systems. We have to address issues arising from the combination of interconnected and often contradictory logical models, such as, e.g., order-preserving/-ignorant, aggregate-oriented/-ignorant, schema-full/-less/-mixed approaches, intra- and inter-model references, intra- and inter-model integrity constraints, or full and partial intra- and inter-model data redundancy. Hence, a number of mature and verified approaches for various data management tasks commonly used for single-model DBMSs cannot be directly applied to multi-model DBMSs.

This thesis aims to propose a new family of unified approaches for both conceptual and logical multi-model modelling and data management. We first analyse the state-of-the-art of related areas. Then we propose abstract data structures to represent multi-model schema and data. These structures are then utilised in the design of approaches for unified schema inference, data migration, schema evolution, and correct backward propagation of changes to the data. All the proposed approaches are implemented and experimentally verified.

Keywords: Multi-Model Data, Conceptual Modelling, Logical Modelling, Schema Inference, Data Migration, Evolution Management, Category Theory

Contents

Preface **5**

Commentary **7**

0.1 Variety of Data . . . 9
0.1.1 Basic Constructs and Their Unification . . . 10
0.1.2 Multi-Model Data . . . 10
0.2 Category Theory . . . 12
0.2.1 Choice of Category Theory . . . 12
0.2.2 Apparent Similarity to Graph Theory . . . 12
0.2.3 Application of Category Theory in the Proposed Approach 13
0.3 Multi-Model Data Modelling . . . 14
0.3.1 Conceptual Layer . . . 14
0.3.2 Logical Layer . . . 20
0.3.3 Open Questions and Challenges in Data Modelling . . . 35
0.3.4 Contribution: Framework MM-cat . . . 38
0.4 Schema Inference . . . 41
0.4.1 State of the art . . . 41
0.4.2 Closely Related Single-Model Approaches . . . 44
0.4.3 Open Questions and Challenges in Schema Inference . . . 47
0.4.4 Contribution: Framework MM-infer . . . 50
0.5 Evolution Management . . . 52
0.5.1 Closely Related Approaches . . . 52
0.5.2 Open Questions and Challenges in Evolution Management 60
0.5.3 Contribution: Framework MM-evocat . . . 62

1 Categorical Management of Multi-Model Data **65**

1.1 Introduction . . . 66
1.2 Categories . . . 67
1.3 Framework . . . 67
1.3.1 Multi-Model Scenario . . . 68
1.3.2 Schema Modeling . . . 69
1.3.3 Database Decomposition . . . 70
1.3.4 Data Representation . . . 71
1.3.5 Query Language . . . 72
1.3.6 Query Evaluation . . . 74
1.3.7 Evolution Management . . . 76
1.4 Summary and Conclusion . . . 76

2 Categorical Modeling of Multi-Model Data: One Model to Rule Them All **79**

2.1 Introduction . . . 80
2.2 Related Work . . . 81
2.3 Preliminary Concepts . . . 82
2.3.1 Category Theory . . . 82
2.3.2 Conceptual Modeling . . . 83

2.4 Schema Representation 83
2.4.1 Schema Category 83
2.4.2 Instance Categories 84
2.4.3 Schema Translation 85
2.5 Categorical Framework 90
2.6 Proposal Evaluation 91
2.7 Conclusion 92

3 A Unified Representation and Transformation of Multi-Model Data using Category Theory 93
3.1 Introduction 94
3.2 Unified View of Multi-Model Data 97
3.2.1 Basic Concepts of Category Theory 99
3.2.2 Categorical Representation of Multi-Model Data 100
3.3 Category-to-Data Mapping 103
3.3.1 Formal Definitions 105
3.3.2 JSON-like Representation 107
3.4 Transformations 110
3.4.1 Model-to-Category Transformation 110
3.4.2 Category-to-Model Transformation 117
3.4.3 Multi-model-to-Multi-model Migration 122
3.5 Framework *MM-cat* 123
3.5.1 Architecture and Implementation 126
3.6 Benefits of Category Theory 132
3.6.1 Application - Querying 133
3.7 Related Work 137
3.8 Conclusion 138
3.8.1 Future Work 139

4 A Universal Approach for Multi-Model Schema Inference 141
4.1 Introduction 142
4.2 Related Work 144
4.3 Data Models and Their Unification 146
4.3.1 Overview of Models 146
4.3.2 Unification of Models 152
4.4 Multi-Model Schema Inference 153
4.4.1 Basic Building Blocks 153
4.4.2 Schema Inference Workflow 157
4.4.3 Record-Based Local Inference Algorithm 160
4.4.4 Property-Based Local Inference Algorithm 162
4.4.5 Gathering Candidates for ICs and Redundancy 166
4.4.6 Global Phase 171
4.5 Architecture and Implementation 174
4.5.1 Database Wrappers 177
4.6 Experiments 181
4.7 Conclusion and Future Work 185

5 MM-evocat: A Tool for Modelling and Evolution Management of Multi-Model Data 187
5.1 Introduction . . . 188
5.2 Related Work . . . 190
5.3 Categorical Conceptual Model . . . 190
5.4 Evolution Manager *MM-evocat* . . . 193
5.4.1 Schema Modification Operations . . . 193
5.5 Demonstration Outline . . . 196

Conclusion 199

Bibliography 201

List of Figures 215

List of Tables 219

List of Algorithms 221

List of Abbreviations 223

List of Publications 227

A Category Theory 229
A.1 Basic Definitions . . . 229
A.2 Functors . . . 232
A.3 Natural Transformations . . . 235
A.4 Universal Constructions . . . 237

Preface

The proposed thesis presents selected results of the author's research in the area of modelling and management of multi-model data. The research has been carried out at the Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague in years 2019-2022. The author is a member of Multi-Model Databases Research Group[1] lead by doc. RNDr. Irena Holubová, Ph.D.

The results are presented as a collection of five selected papers [1, 2, 3, 4, 5] followed by a unifying commentary. Paper I is a vision of a categorical framework, Paper II addresses conceptual modelling of multi-model data, Paper III proposes unifying data structures together with universal schema and data transformation algorithms, Paper IV deals with the inference of a unifying schema from already existing data, and Paper V addresses the problem of schema evolution and its backwards propagation. A complete list of 14 papers – namely, 2 Q1 academic journal articles (2x Journal of Big Data), 3 CORE A conference papers (EDBT 2022, MODELS 2022, MODELS 2021), 5 CORE B conference papers (IDEAS 2022, SAC 2022, ENASE 2022, 2x IDEAS 2021), 1 CORE C conference paper (MEDI 2021), 2 workshop papers (PhD@DASFAA 2021, CoMoNoS@ER 2020), and 1 manuscript under review – is provided at the end of this thesis.

Prior to the summary of the papers, a commentary is provided for each addressed area, namely a motivation, a brief summary of the state-of-the-art, a list of open questions and challenges, and a discussion of our contribution. For convenience, the references to the author's original contribution are marked with a pictogram (see on the right). Finally, we conclude and outline directions of our current and future research. ★

The research included in the selected papers has been supported by several grants, namely the GAČR project no. 20-22276S, and the project GA UK no. 16222 (principal researcher).

Želivec, July 2022

Pavel Koupil (Čontoš)

[1] https://www.ksi.mff.cuni.cz/area.html?id=multi-model-data

Commentary

For decades, relational database management systems (RDBMS) based on the relational model [6] were often the obvious candidate for data management. These robust and time-verified systems were characterised by a schema-first, data-later approach and handled structured data very well. However, nowadays, most of the data consists of very large (***volume***) and varied (***variety***) (un)structured data, which in addition are rapidly generated (***velocity***) and changed (***variability***). Hence, relational databases do not necessarily meet the new data management requirements.

Around the dawn of the second millennium, a new family, the so-called NoSQL database systems [7], has emerged, pushing the boundaries of many approaches to data processing. Compared to the RDBMSs, these systems work much better with complex (un)structured data and respond to changes in data and user requirements, e.g., due to the absence of an explicit schema (data first, schema later/never approach). In addition, the NoSQL systems are often scalable, i.e., they allow us to respond flexibly to the volume of data being processed. Despite all the advantages of NoSQL systems, they are not a replacement for RDBMS, but the two families complement each other appropriately.

However, even the advent of NoSQL systems has not solved all the problems. Currently, one of the most difficult challenges is the ***variety*** of data, i.e., a large number of various data types and formats. For example, based on the structure, data can be classified as structured, semi-structured, and unstructured, and/or based on logical representation, there exists, e.g., *relational*, *array*, *graph*, *key/value*, *document*, and *columnar* data. Besides, in real-world applications the logical models are often combined, overlapped, and linked by references. Hence, the applications deal with so-called *multi-model data*.

In general, approaches that store and process multi-model data can be divided into two groups. The first group consists of (mainly) academia-driven systems, the so-called *polystores* [8], which are based on the idea of *polyglot persistence* [7]. These are a combination of single-model database systems that are managed by a so-called *mediator*, which allows for the use of a single interface. To name just a few representatives, there is, e.g., BigDAWG [9] or Estocada [10]. Alternatively, there exist industry-driven so-called *multi-model database management systems* [11], which support multiple logical models within a single system, where all models are treated as first-class citizens [12]. Obviously, this provides a single interface to work with the data. Currently, there are dozens of representatives of multi-model databases,[2] including originally single-model systems now supporting additional data models [13] or attempts to natively implement multiple data models, such as, e.g., Octopus[3] and ArangoDB.[4]

Although there exists a number of mature approaches for various data management tasks commonly used for single-model DBMSs, most of them cannot be directly applied to multi-model DBMSs. The aspect of multi-model data introduces a new dimension of complexity and new challenges not seen in single-model

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

[3] https://octopus.com/docs/administration/data/octopus-database

[4] https://www.arangodb.com

systems. We need to address issues arising from the representation of data by a single logical model, as well as from the combination of interconnected and often contradictory models, e.g., (cross-model) references, full and partial (cross-model) data redundancy, and (cross-model) integrity constraints. In general, we lack a family of approaches that focus on:

- *Grasping the contradictory features of different data models.* Ideally, we need a unified (abstract) conceptual representation of the data models that hides the minor differences and puts the corresponding features of the logical models on the same level, allowing us to work with them in a unified way [14].

- *Mutual mapping of conceptual and logical layers.* Having a unified conceptual layer, we need a way to transform this layer into a logical layer. Currently, there exist approaches that transform the conceptual layer into, e.g., a relational model [15]. However, in the case of a combination of multiple logical models, this approach is not straightforward at all, mainly due to the different schema approaches (i.e., schema-full, schema-mixed, and schema-less) and, again, the contradictory features of logical models.

- *Inference of the multi-model schema.* To handle schema-mixed and schema-less data, there exist multiple approaches that infer the implicit schema from already stored data [16]. However, to the best of our knowledge, none of them is generally applicable to multi-model data, i.e., one cannot infer features arising from the combination of multiple models.

- *Unified query language.* Currently, there are many, often non-standardised query languages (not only) for multi-model systems [13], which create a huge burden for the users. The ideal situation is the existence of a single universal and natural language whose expressive power embraces commonly used query constructs and which allows efficient querying over multiple interconnected models.

- *Evolution management and correct propagation of changes.* As user requirements change, the data structures evolve. Hence, we need an approach that is universally applicable to propagate changes to all affected parts of the multi-model system correctly and completely. Moreover, changes in schema and data representation can be exploited to increase the performance of, e.g., querying and other data tasks.

Outline The rest of the thesis is structured as follows: In this Chapter we give an introduction to the selected problems, an overview of open questions and we provide a commentary on our proposed solutions. In Chapter 1 we discuss the vision of a whole multi-model framework. In Chapter 2, we detail our proposed approach to abstract modelling of multi-model data. In Chapter 3 we propose a family of algorithms for data transformation that are independent of the logical representation of the data. In Chapter 4 we describe the proposed algorithm for inference of multi-model schemas. In Chapter 5 we present tools for multi-model schema and data evolution. Finally, we conclude and outline future work.

0.1 Variety of Data

Besides relational databases representing data as relations, NoSQL databases allow us to represent and store data using other models, e.g., as a hierarchical tree data (and structures) or pure graph data. We can classify data models as aggregate-ignorant and aggregate-oriented.

The traditional representative of aggregate-ignorant models is the relational model. This group also includes the array model, which allows to represent spatial data, however with certain limitations (e.g. references between arrays are not supported). An example of a system that implements the array model is, e.g., SciDB.[5] Another aggregate-ignorant representative is the graph model, implemented, e.g., in Neo4j,[6] which allows to represent data in its natural form, i.e., as a system of connected related real-world objects. These connections then allow a completely different querying principle (e.g., graph traversal, neighbourhood search etc.) compared to SQL and its derivatives. Similarly, the RDF model corresponds to a directed graph composed of triple statements.

The simplest representative of aggregate-oriented models is the key/value model implemented in, e.g., Redis[7] and Riak[8] systems. Here, the data is stored as a pair $(key, value)$, with *key* (identifier) referring to *value*, i.e., an object stored in the database as a black box. The document model, implemented in, e.g., MongoDB[9] or MarkLogic,[10] uses a similar principle, i.e., it is also based on pairs $(key, value)$, however, the pairs may form a hierarchical structure (i.e., nesting of pairs is allowed). In particular we refer to the pairs as (unordered) *fields* (JSON) or (ordered) *elements* (XML). Unlike the key/value model, querying and referencing over nested data is allowed. Finally, the column model, implemented, e.g., in Apache Cassandra[11] and Apache HBase,[12] also allows related data to be stored together, but in the form of (optional and possibly structured) pairs $(name, value)$ (i.e., columns) forming rows of column families.

Example 0.1. Figure 0.1 illustrates examples of selected data models. The relational table Customer (purple) represents customers together with their contact details, while the graph model (blue) represents the relationships between customers, i.e., a social network. The key/value pairs (yellow) represent the shopping carts of customers. The document model (green) represents the collection of orders of each customer as a hierarchical document. Finally, the column family Orders (red) represents a list of orders for each customer. Thus, at first sight, we can see that data from a single domain can be suitably represented by different logical models. □

Note that comprehensive descriptions of the data models along with examples are provided in Section 4.3. ★

[5] https://www.paradigm4.com
[6] https://neo4j.com
[7] https://redis.io
[8] https://riak.com/products/riak-kv/index.html
[9] https://www.mongodb.com
[10] https://www.marklogic.com/product/marklogic-database-overview/
[11] https://cassandra.apache.org/_/index.html
[12] https://hbase.apache.org

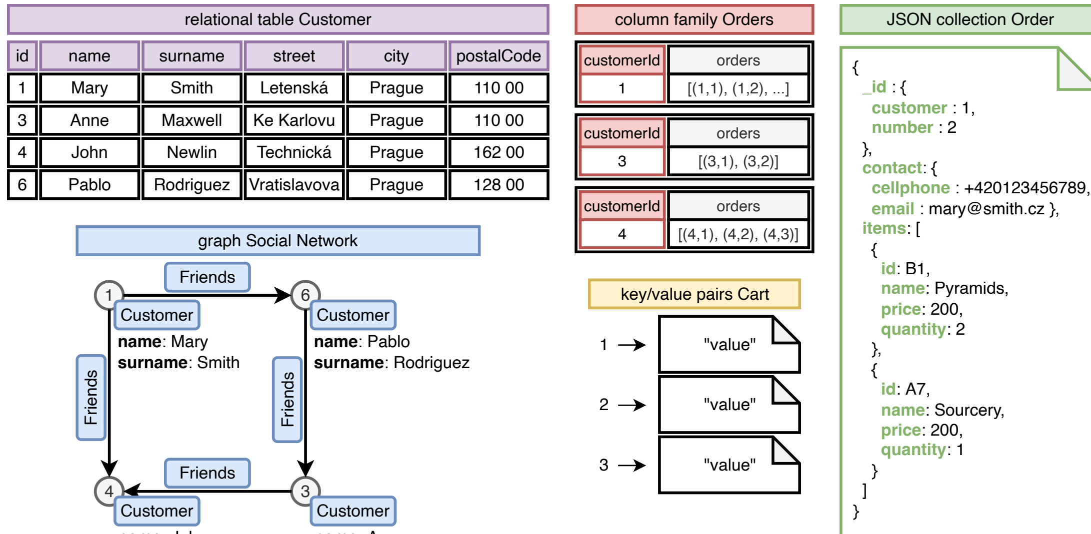


Figure 0.1: An example of variety of data

0.1.1 Basic Constructs and Their Unification

Since the terminology for the constructs of the data models varies greatly, in the following text we will refer to the uniform terminology in Table 0.1. A *kind* refers to a single collection of (possibly similar) instances, corresponding to, e.g., a relational table, a node label, or a collection of documents. A *record* then denotes a single instance of its kind, e.g., a tuple in a relational table, a particular node, or a single document. A record further consists of *properties* such as:

- A *simple property*, e.g., a scalar value.
- A *complex property*, represented, e.g., by a homogeneous[13] or heterogeneous[14] *array*, or a *structure* that contains other properties (possibly both simple and complex).

In addition, complex properties, such as nested documents, form a hierarchy of properties. Hence, a record can be considered as a special kind of a complex root property.

The *domains* correspond to the data types of the *values* of individual properties, while the *active domain* is a set of actively used values. An *identifier* (further distinguished as *simple*, *complex*, and *overlapping*) then uniquely identifies a specific record of a kind. Finally, a *reference* from one kind to an identifier of another kind allows related data to be associated. Note that references are only allowed for certain models.

0.1.2 Multi-Model Data

In general, *multi-model data* is represented by multiple logical models within a single system. Multi-model data not only adopt the properties of single-model data, but in addition:

[13] An array that contains elements of the same type.

[14] An array that contains elements of multiple types. This form of an array is allowed, e.g., in MongoDB document model.

Table 0.1: 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	JSON Field / XML element or attribute	Column
Array	–	–	Array	–	Array	JSON array / repeating XML elements	Array
Structure	–	–	–	–	Set / ZSet / Hash	Nested document	Super 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	JSON identifier / XML ID or key	Row key
Reference	Foreign key	–	–	–	–	JSON reference / XML keyref	–

- Analogous to possibly hierarchical models (e.g., the document model), we can connect multi-model data by (1) *inter-model references* or (2) *inter-model embedding* (e.g., a JSONB column in a PostgreSQL table embeds a JSON document into a relational table).
- Similarly to the property labelled graph in Neo4j, we can express *cross-model redundancy*. In this case, we represent the same parts of the data using a combination of data models. We speak about *partial redundancy* if only a subset of the data is represented by multiple models, or a *complete redundancy* if the entire set of data is represented by two or more data models.

The combination of data models within a larger unit (a polystore or a multi-model database) allows us to use the right tool (data model) for the specific tasks. For instance, we represent structured data with small differences in the document model, data containing a large number of relationships between entities with the need for efficient querying over the relationships between entities in the graph model, or fast generated data without the need for complex querying in the key/value model.

Example 0.2. Figure 0.1 also illustrates multi-model data. Compared to Example 0.1, note that there are cross-model references in the data, e.g. from collection *Order* (*customerId*) to table *Customer* (*id*). The data is also redundant, i.e., customer information is stored in both the relational table and the graph. □

0.2 Category Theory

Category theory [17] is a branch of mathematics that provides a way to generalise mathematical structures and the relationships between them. Hence, it is a unifying theory that is useful for finding connections between different areas, not only in mathematics and theoretical computer science.

In this section, we clarify the choice of category theory, indicate the (apparent) similarity between category theory and graph theory, and explain how category theory is applied in the proposed approaches to modelling and managing multi-model data. Note that the basic definitions underlying our proposal, including illustrative examples that are closely related to real-world applications in data modelling approaches, are provided in the Appendix A.

0.2.1 Choice of Category Theory

We have chosen category theory for the unified representation of multi-model data, because it allows for different levels of abstraction and unifies different types of tasks at the abstract level in a natural way. Therefore it has been successfully applied not only for single-model data modelling [18, 19, 20, 21], but also in various (related) areas, such as programming language theory [22, 23], data migration [20], or artificial intelligence (AI) [24, 25] among others. Thus, we do not need to apply a variety of theories and approaches. Instead, combining concepts such as category (see Definition 1), functor (see Definition 6), or natural transformation (see Definition 10) suitably, we are able to represent, e.g., the conceptual and logical schema, the relationship between these schemas, data instances, querying based on pattern matching, data migration, and evolution management.

0.2.2 Apparent Similarity to Graph Theory

At first sight, category theory is similar to graph theory. That is, both categories and (directed) graphs are usually visualised using points and arrows. However, this is where the similarity ends.

A *category* consists of a collection (class) of *objects* and a collection of *morphisms*, where each morphism associates two objects. In addition, (1) morphisms carry a particular meaning, e.g., they define relations or functions between objects, (2) each object is equipped with a so-called *identity* morphism, (3) morphisms are composable using a *composition* operation for which the associative and transitive laws hold, and (4) morphisms and their composition can be compared with each other. Hence, one cannot arbitrarily orient morphisms in a category, but must always respect the composition operation. Finally, categories can, e.g., be mapped, transformed and translated to each other using the notion of *functor*.

In contrast, a *graph* consists of a set of *vertices* (i.e. a special case of a collection) and a set of *edges*, where each edge connects two vertices. Furthermore, edges do not carry any additional meaning[15] and no operations are defined to

[15] However, if edges carry meaning, e.g., a cost, then this is an extended definition of the general graph and its application.

allow edges to be composed or compared. We consider only the notion of a *path* in the graph. In other words, a graph only describes a structure. Finally, graphs can be compared with each other using the notion of *graph homomorphisms*.

However, note that each graph generates a category, referred to as a *free category* (see Definition 5) also known as a *path category*, in which the vertices of the graph form objects and the paths in the graph form morphisms (see Example A.3). In addition, each free category is a *small category* (see Definition 3) and each small category has an underlying graph.

0.2.3 Application of Category Theory in the Proposed Approach

We applied category theory in our approach to multi-model schema and data representation (see Chapter 2), data migration (see Chapter 3), and schema and data evolution (see Chapter 5). ★ ★

The main objective while proposing our approach was to make it user-friendly. Thus, on the one hand, we utilise a complex unifying theory, but on the other hand, we deliberately exploit the apparent similarities with graph theory. For example, we represent the structure of data as a graph (i.e., vertices represent classes of real-world objects, edges represent links between these classes) that freely generates a schema category.

Moreover, to avoid unnecessarily burdening the reader with advanced category theory constructs, e.g., natural transformation (see Definition 10) or universal constructions (see Definitions 13, 14, 16, and 17), we consider these constructs implicitly in our proposal. For example, instead of explicitly using the notions *product* and *coproduct* in the case of the representation of an identifier (i.e., a special case of a *product*) and a set of (overlapping) identifiers (i.e., a coproduct of identifiers), we introduce an internal object (or graph vertex) representation that consists of common notions of a superidentifier and a set of identifiers.

Hence we believe that the model we propose is simple enough that anyone with a basic knowledge of category theory, i.e., the definition of a category (see Definition 1), and functor (see Definition 6),[16] will find our approach easy-to-use.

Although it may seem that we only apply graph theory, we still (implicitly) exploit various levels of abstraction and advanced category theory constructs on which we base the representation of instance data (e.g., functors or natural transformations), data migration (e.g., functors, universal constructions, or natural transformations), schema modification (e.g., functors) and, as future work, querying (e.g., universal constructions or natural transformations).

For the convenience of the reader, all the cases of application of category theory in our approach are summarised in the commentary on data modelling (see Subsection 0.3.4) and the commentary on schema evolution and data migration (see Subsection 0.5.3). Finally, in the Appendix A we also outline in which approaches and for which purpose the above definitions are applied, i.e., we provide additional examples.

[16] Or with a basic knowledge of graph theory, i.e., the definition of a graph and a graph homomorphism.

0.3 Multi-Model Data Modelling

The objective of data modelling is a mapping of real-world objects and their structures to data objects and relationships between them. The conceptual layer captures a generally applicable and platform-independent model of a part of reality. The logical layer is a platform-specific representation of logical data structures in particular (database) systems. Finally, the physical layer organises data into physical units and addresses, e.g., data access.

Currently, there exist a number of approaches for modelling at the conceptual (see Subsection 0.3.1) and logical (see Subsection 0.3.2) layer. However, these are approaches proposed with a relational or graph model in mind as we analyse ★ in [26]. Hence, their general applicability to multi-model data is limited, as the approaches are often unable to capture new structural properties in the data, such as, e.g., relations between properties (note that ER and UML only allow to capture relationships between classes of objects).

In addition, the variety of data models at the logical layer allows to represent data in different ways. However, due to contradictory features of data models, a change in logical representation is not straightforward. Both schema and data loss may occur during this process, e.g. when an order of structural elements carries some information. There have been attempts to unify logical data models [19, 27, 28, 29, 30], but these are often suboptimal solutions that, moreover, cover only a limited subset of existing data models. Although attempts in unification have been made, usually model-specific constructs are inherited [30], thus a broader applicability is still limited.

In the following subsections, we discuss selected existing approaches to modelling at the conceptual and logical level and demonstrate how and if these approaches can be used to represent multi-model data. At the logical level, we mainly focus on approaches that attempt to abstract data models. We compare the selected solutions and based on their analysis we discuss a set of open questions and challenges. Finally, we present a novel multi-model data modelling approach. (Note that in this section we only discuss a static model. Its changes are addressed in a separate section 0.5.)

0.3.1 Conceptual Layer

The objective of conceptual modelling is to represent data without being bound by the features of particular logical models, i.e., we speak about so-called *platform independent modelling* (PIM).

To achieve such a unified abstraction, traditional conceptual modelling languages, e.g., the *Entity-Relationship* model (ER) [31] and the *Unified Modelling Language* (UML) [32] (namely class diagram), suffice with the notions of *entity*, *relation*, *attribute*, *identifier*, and *multiplicity*. Additionally, there also exist approaches [18, 33] based on category theory, or currently less widely used approaches such as the *Natural language Information Analysis Method* (NIAM) [34], representatives of the *Fact-Oriented Modelling* (FORM) [35, 36], or representatives of the *Formal Semantic Database Modelling* [37], e.g., the *Functional Data Model* (FDM) [38, 39], the *Semantic Data Model* (SDM) [40], and the *IFO Model* (IFO) [41]. Last but not least, there also exists an approach [42] allowing to

represent document and graph data at the conceptual layer.

Taking the best of their features, we have proposed a multi-model schema and data abstraction approach [2] inspired by ER and UML and based on category theory. Hence, in this subsection, we mainly discuss the first three mentioned conceptual modelling approaches.

Entity-Relationship Model

The first representative that enables conceptual modelling of complex structures, i.e., real-world entities including their attributes and mutual relationships, is the ER language. The basic constructs are as follows:

- An *entity type* reflects a class of real-world objects. It must contain an identifier and (optionally) includes also additional attributes.
- A *relationship type* represents a binary, n-ary, or reflexive connection between classes of objects. It is implicitly identified by the participants in the relationship, and thus explicit identifier is not allowed. However, similarly to the entity type, the relationship type may contain additional attributes. Moreover, the ER language also allows to name individual roles in a relationship.
- The so-called *weak entity type* is (co-)identified by all other participants of a selected relationship. Note that the ER language lacks the ability to include only selected participants in the weak identifier.
- The *ISA hierarchy* is a possibility to express generalisation (ancestor) or specialisation (descendant) of entity types.
- An *attribute* expresses a characteristic of a class of real-world objects or their relationships in the form of a required, optional, multi-valued, or structured attribute.
- An *identifier* is a special type of an attribute. As such it has identification functionality within the same class of real-world objects. Moreover, identifiers can be categorised based on two criteria:
  - In terms of its structure, we distinguish between a simple identifier (consisting of a single attribute), a composite identifier (consisting of multiple attributes), and an overlapping identifier (i.e., there is at least one attribute that is part of two different identifiers).
  - In terms of entity type membership, a strong identifier (i.e., being directly an attribute of the entity type), an inherited identifier (i.e., a member of the ancestor entity within the ISA hierarchy), a weak mixed identifier (i.e., the entity type is partially identified by an identifier of another entity type with which it enters into a relationship), or a weak external identifier (the entity type is fully identified by an identifier of related entity type) can be distinguished.
- A structured attribute is another special type of a (hierarchical) attribute that can have only trivial depth of 1 (i.e., no further nesting of attributes is allowed).

- A *cardinality*, described as a pair (min,max), $min \in \{0, 1\}$, $max \in \{1, *\}$, whereas $min \leq max$, expresses multiplicities between an entity type and a relationship type (within a relationship) or between an entity type and its attribute.

The ER language exists in several distinct notations [43], e.g., Chen [31], Reiner et al. [44], Teorey [45], Hoffer et al. [46], and IDEF1X [47], that mutually differ not only visually but also in the constructs used. In other words, there is no standardised format for this language.

Example 0.3. Figure 0.2 illustrates the ER diagram of multi-model data from Figure 0.1. At first glance, the diagram looks complete, i.e., faithfully representing all the features of the data at the conceptual level. For example, there is an entity type *Customer* having two identifiers (*id*) and (*name, surname*), a weak entity type *Order* having a mixed identifier (*id, number*), and the ISA hierarchy between *Product* and its children *Audiobook* and *Book*. Unfortunately, the ER language only allows us to model traditional structured attributes (e.g., *Address*), whereas *Contact* which can be understood as a structured attribute composed of pairs (*name, value*), can only be represented as a binary relationship between *Order* and *Type*. □

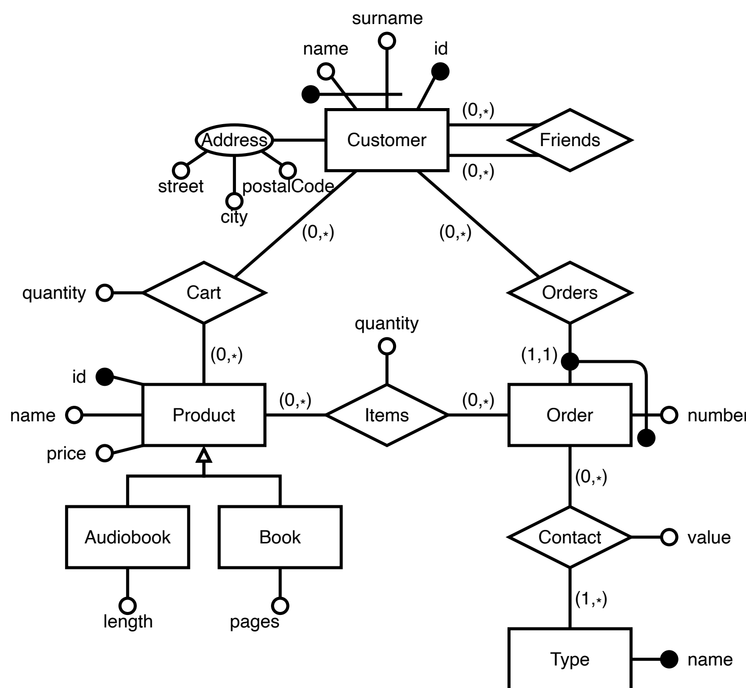


Figure 0.2: An example of ER schema

Unified Modeling Language

Alternatively, the standardised UML allows us to visually represent not only complex structures, but also entire systems. Using the UML, a number of diagrams can be created (i.e., so-called *behavioural* and *structural* diagrams), of which a representative of structural diagrams, so-called *class diagram*, can be used to model the conceptual schema.

The basic constructs of the class diagram are as follows:

- A *concept* is a named set of attributes that represents a class of real-world objects. It corresponds to an entity type in the ER language.

- An *association* is a named relationship (connection) between concepts. It can be defined between two (binary) or more (n-ary) concepts, but also over a single concept (reflexive). In addition, similarly to the ER language, an *association class* can be used to represent the connection between concepts.

- An *attribute* is a logical and untyped data value of a concept or an association. Note that a structured attribute is not an explicit part of the UML, but it can be expressed as an additional concept attached by an association.

- *Multiplicity* expresses the number of instances of concept A associable with an instance of concept B.

- Finally, *inheritance* allows the expression of generalisation (ancestor) or specialisation (descendant) of concepts. Similarly to ER, also *multiple inheritance* is allowed.

The expressive power of the class diagram is limited and does not cover all the important details of the conceptual schema, such as, e.g., identifiers or weak entity types.

Example 0.4. Figure 0.3 illustrates the UML class diagram corresponding to schema of the multi-model data from Figure 0.1. Since UML does not provide a graphical distinction between identifiers and attributes, the identifiers identifying the *Customer* concept cannot be visualised. Moreover, the concept *Order* is not considered as a weak concept. Also note that the structured attribute *Address* is represented as a separate concept that is linked to the parent concept by an association. Furthermore, *Contact* is also represented by an association between the concepts *Order* and *Type*, similarly to the ER model in Figure 0.2. The class diagram also allows us to represent inheritance, specifically the concepts *Audiobook* and *Book* are descendants of the concept *Product*. □

Categorical Conceptual Model (Lippe and Ter Hofstede)

Last but not least, the approach [18] allow to model the conceptual schema and data using an approach based on category theory. The foundation of the approach is a directed multi-graph, so-called *type graph*, that freely generates a category $\mathbf{C}$ (see Definition 5) representing the conceptual schema.

The *type graph* $G = (V, E, L, lbl, pow)$ is a tuple consisting of:

- A set of vertices V, where each vertex $v \in V$ represents a particular type of real-world objects, a relationship type, or an attribute type.

- A set of edges E, where each edge $e \in E$ is an (optionally labelled) directed pair of vertices $e : v_1 \rightarrow v_2$, $v_1, v_2 \in V$ determining the way how the vertices participate in various constructions.

- A set of labels $L := \{role, spec, gen, power_role, elt_role\}$, where $role$ represents a connection between a relationship type and its participant type,

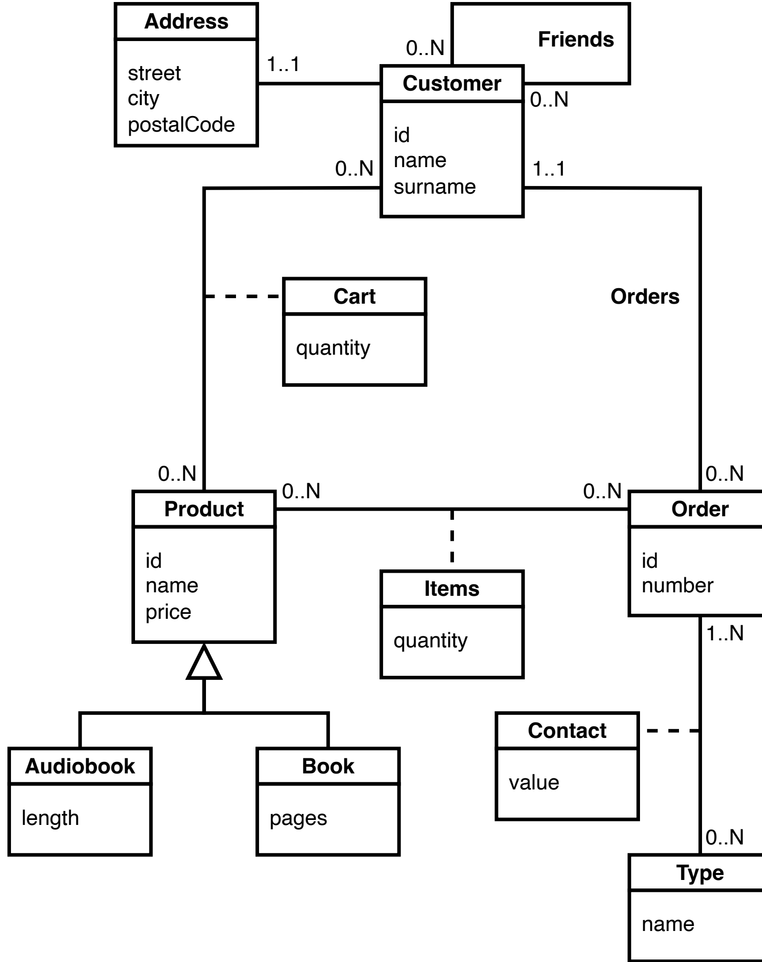


Figure 0.3: An example of UML schema

spec represents a type specialisation, *gen* represents a type generalisation, and *power_role* and *elt_role* representing participants in so-called power type (i.e., a concept of a multi-valued property, e.g., set), the former one representing a connection between a power type and a parent type and the latter one representing the connection between a power type and an element type.

- Function $lbl : E \to L$ associating an edge $e \in E$ with a label $l \in L$.
- Function *pow* is a bijection from edges with label *power_role* to edges with label *elt_role*, which says that an instance of a power type can be identified if and only if its elements are identifiable.
- It must hold that no cycles in the graph are composed of edges with labels *spec* or *gen*.

Moreover, in the conceptual schema $\mathbf{C} = \{\mathcal{O}_{\mathbf{C}}, \mathcal{M}_{\mathbf{C}}, \circ, 1\}$ the following holds:

- The uniqueness of an attribute is represented by *monomorphism* $mono \in \mathcal{M}_{\mathbf{C}}$ (see Definition 4), i.e., each element of $cod(mono)$ determines at most one element in $dom(mono)$.

- A type of a complex identifier is expressed as a product with projections (see Definition 13) to the components of the identifier (see Example 0.5). Moreover, there is a single monomorphism $mono : P \to I$ between an object $P \in \mathcal{O}_{\mathbf{C}}$ corresponding to type of real-world objects $v_p \in V$ and object $I \in \mathcal{O}_{\mathbf{C}}$ corresponds to type of a complex identifier $v_i \in V$.

- A type of a structured attribute is represented similarly to a type of a complex identifier. The only difference is that there is an epimorphism (see Definition 4) $epi : P \to A$ between an object $P \in \mathcal{O}_{\mathbf{C}}$ corresponding to type $v_p \in V$ and an object $A \in \mathcal{O}_{\mathbf{C}}$ corresponds to type of the structured attribute $v_a \in V$. In other words, each value of a structured attribute must be a part of an instance of the parent type.

- The multiplicity of an attribute is expressed as the product $P \times E$ (i.e., a power type), where an object $P \in \mathcal{O}_{\mathbf{C}}$ corresponding to the parent type $v_p \in V$ and an object $E \in \mathcal{O}_{\mathbf{C}}$ corresponds to element type $v_e \in V$. Moreover, it must hold that both projections $\pi_1 : P \times E \to P$, $\pi_2 : P \times E \to E$ are epimorphisms. Note that this approach can be applied to modelling of, e.g., sets, but it is not applicable to represent data collections in general, e.g., arrays (a collection of ordered and possibly duplicate elements), and maps (a sets of pairs (*name, value*)) distinguishable by *name*).

- Inheritance is represented by a complementable monomorphism (see Definition 15) corresponding to an edge with a *spec* label. In other words, each instance of a child must correspond to a unique instance in each of its ancestors. Also note that multiple inheritance is allowed. Moreover, the subtype diagram commutes, i.e., the children have an access to attributes of theirs ancestors.

- Generalisation of objects $A, B \in \mathcal{O}_{\mathbf{C}}$ is represented as pushout $A + B$ (see Definition 17).

Finally, the approach also allows to represent data instances conceptually. The foundation for data representation is the so-called *instance category*, which can be based on the categories of sets **Set**, finite sets **FinSet**, partial sets **PartSet** (allowing to represent missing values, i.e., null), relations **Rel**, etc. [18].

Example 0.5. Figure 0.4 illustrates a conceptual categorical schema of the multi-model data from Figure 0.1 represented using approach [18]. For the sake of clarity, we do not show the identity morphisms and we divided the schema into two parts: Figure 0.4 (a) depicts only types of real-world objects and their relationships. Figure 0.4 (b) depicts the attributes of type *Customer*.

Note that relationships (i.e., *Friends*, *Items*, *Cart*, *Orders*, and *Contact*) are modelled as a product with *role*-labelled projections to the participants of the relationship. On the other hand, inheritance is modelled as a co-product with *spec*-labelled inclusions (i.e., the child is a specialisation of the parent). Figure 0.4 (b) illustrates the representation of identifiers using monomorphisms (explicitly labelled with an existence quantifier). In other words, there is only a single mapping between type *Customer* and its *id*. The complex identifier (*name, surname*) is represented as a product with projections to attributes *name* and *surname*. There

is once again a single mapping between *Customer* and the complex identifier. Finally, a structured attribute is represented as a product with projections, but in this case $Address : Customer \rightarrow String \times String \times String$ is an epimorphism. Also note that composition $street \circ Address$ allows direct access from *Customer* to *String*. □

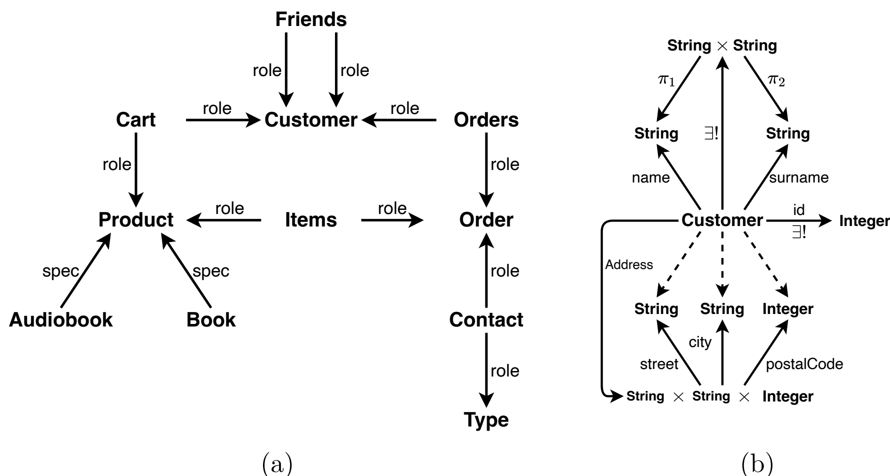


Figure 0.4: An example of a conceptual schema (Lippe and Ter Hofstede)

Comparative Summary

Table 0.2 summarises the expressive power of the three described approaches. In addition, it involves also a comparison of our approach inspired by them and ★ described in detail in Chapter 2.

In principle, all the presented approaches allow for modelling of the traditional concepts of real-world objects, relationships, and their attributes. They differ mainly in their ability to model identifiers, where the UML class diagram does not allow to specify an identifier[17] and, therefore, it does not even work with the principle of a weak entity type partially or completely identified by another entity type. Note that in ER we can model an identifier as a special type of an attribute and that in the categorical approach we use monomorphisms. Moreover, the categorical approach considers an explicit identifier of a relationship type, whereas in the ER language the relationship type is identified implicitly by the participants of the relationship. Finally, the selected approaches differ in the possibility of expressing a structured attribute. The categorical approach allows expressing a structured attribute with unlimited depth and unrestricted structure, while the ER language allows only structured attributes of trivial depth and the UML class diagram expresses structured attributes using a concept and an association.

0.3.2 Logical Layer

While at the conceptual level we view the data in a platform-independent way, the objective of so-called *platform-specific modelling* (PSM) is to capture often

[17]UML allows us to textually express integrity constraints, including the identifier, using the Object Constraint Language (OCL) [48].

Table 0.2: Expressive power of approaches modelling conceptual layer

	ER [31]	**Class diagram (UML)** [32]	**Lippe and Ter Hofstede** [18]	**MM-cat** [3]
Object class	Entity type	Concept	Object	Object
Relationship class	Relationship type	Association (class)	Product + projections (role)	Product + projections
Simple attribute (property)	Attribute	Attribute	Epimorphism to object	Epimorphism to object
Map-like property	No	No	No	Product + projections
Multiplicity	Cardinality (Chen)	Multiplicity	Product + projections (power_role, elt_role)	Product + projections
Role	Role	Named association	No	No
Inheritance	ISA hierarchy	Inheritance	Morphism (*spec*)	Object + monomorphism
Generalisation	Multiple ISA hierarchy	Multiple inheritance	Injection (*gen*)	Injection
Structured attribute (property)	Structured attribute	Concept + association	Product + projections	Product + projections
Reflexive relationship class	Reflexive relationship type	Reflexive association	Object + morphisms (*role*)	Object + morphisms
N-ary relationship class	N-ary relationship type	N-ary association (class)	Product + morphisms (role)	Product + projections
Weak object class	Weak entity type	No	Object + monomorphism	Object + monomorphism
Identifier	Identifier	No	Monomorphism	Monomorphism
Complex identifier	Complex identifier	No	Monomorphism to product	Monomorphism to product
Multiple identifiers	Yes	No	Yes	Coproduct of products
Overlapping identifier	Overlapping identifier	No	No	Overlapping products
Relationship identifier	Implicit	No	Implicit	Implicit / Explicit
Integrity constraints	Identifier	OCL	Identifier	Identifier, Reference

non-transferable characteristics of particular database systems. For example, we consider particular data structures, i.e. graphs, trees, and matrices implemented by the actual DBMS representatives,[18] as well as academic proposals (e.g., the X-SEM model [49] for XML data).

In this subsection, we discuss selected existing approaches that address the unification of different logical models, in particular category theory based approaches [19, 20, 21], the NoSQL abstract model (NoAM) [27], associative arrays (AA) [28], the tensor data model (TDM) [29], and the U-Schema [30]. We also verify if the approaches are generic enough to cover various characteristics of popular data models and multi-model data in general.

[18]Various popular models based on data structures are discussed in detail in Section 4.3. ★

Categorical Graph-Oriented Object Data Model (CGOOD)

The approach [19], which is based on the *Graph-Oriented Object Database Model* (GOOD) [50], enables us to work with object and relational data in a unified way (i.e., it aims to abstract these two approaches at the logical level). Abstract data instances are represented as typed graphs, with schema and data defined solely in terms of categorical constructs. In addition, this approach allows graph pattern matching, which further provides a basis for evolution management and querying.

The core structure of the CGOOD model is the directed graph $G = (V, E, srt, tgt)$, which represents both the schema (i.e., so-called *typegraph*) and the data. An object (from the object model), a corresponding tuple (from the relational model), or an active domain of a property is represented by a vertex $v \in V$, whereas the property (i.e., the fact that a certain value is a property of a complex object) is represented by an edge $e \in E$. Note that an edge also represents a function between two types of objects, e.g., *getter*, *isAncestor* (isa), etc. Finally, functions $src, tgt : E \to V$ assign the source and target vertices to the edge accordingly.

Categorically speaking, a particular graph G is represented as a set-valued functor from Example A.5. Moreover, a collection of such graphs forms a category of all graphs $\mathbf{G} = (\mathcal{O}_\mathbf{G}, \mathcal{M}_\mathbf{G}, \circ, 1)$, which corresponds to the functor category from Example A.8. The category $\mathbf{G}$ also provides a fundamental framework for the definition of a data instance. A so-called *typed instance* is a morphism $Inst : G \to T$, where $Inst \in \mathcal{M}_\mathbf{G}$, and $G, T \in \mathcal{O}_\mathbf{G}$. Note that $Inst$ corresponds to a graph homomorphism between G and T, i.e., there exists a mapping $Inst_V : V_G \to V_T$ specifying the type of the value, and a mapping $Inst_E : E_G \to E_T$ preserving the structure of G.

Example 0.6. Figure 0.5 (a) illustrates a typegraph representing the logical schema of the multi-model data from Figure 0.1. Figure 0.5 (b) depicts a part of the data. For easier understanding, the vertices and edges of the typegraph are labelled by strings, though this is not necessary from a categorical perspective (CGOOD works with unlabelled graphs). In addition, we use colours to represent the mapping of the data to the corresponding types, i.e., values 1, 3, 4, 6 are mapped to *Id*, values Mary, Anne, John, Pablo are mapped to Name, etc. Note that the mapping preserves the structure, i.e., if there exists an edge between two vertices in the data, then there will be an edge between the corresponding vertices in the typegraph. Finally, note that the data does not contain explicit identifiers. □

To conclude, note that category theory in the context of relational and object-relational models has also been addressed in the works [18, 51, 52, 53].

Categorical Logical Model (Spivak et al.)

The approach [20] represents the relational database schema as a small category (see Definition 3) and corresponding data instance as a functor (see Definition 6). The authors also propose a category of all schemas and allowed operations between the schemas. Hence, in combination with so-called *data migration functors* built on top of the schema operations, the migration of data instances is allowed. Moreover, these data migration functors form the basis of a categorical query lan-

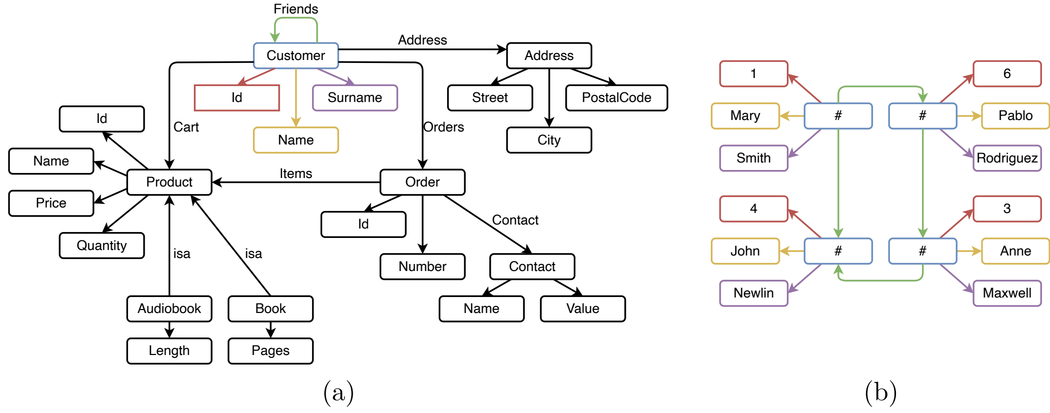


Figure 0.5: An example of CGOOD application to multi-model data

guage CQL [54][19] over relational data. Finally, this approach allows to convert relational data to RDF triples [55] and vice versa.

However, in order to apply this approach, the authors presume that the database (i.e., a set of named kinds) is in the so-called *categorical normal form*, defined as follows:

- Each kind t contains a simple identifier id_t.
- For each additional property c of kind t, $c \neq id_t$, there is a target kind t' such that each record of kind t is mapped to exactly one record of type t', i.e., there is a mapping $c : t \to t'$.
- If two paths $p : t \to t'$, $q : t \to t'$, $p \neq q$ represent the same mapping of records between two kinds, the equivalence $p \simeq q$ must be included in the schema.

In addition, the authors define the notion of a categorical path equivalence relation (CPER) on the graph $G = (V, E, src, tgt)$ denoted by $\simeq$, which has the following properties:

- If p,q are paths of the graph G and $p \simeq q$, then $src(p) = src(q)$.
- If p,q are paths of the graph G and $p \simeq q$, then $tgt(p) = tgt(q)$.
- Let $p, q : b \to c$ be paths and $m : a \to b$ be an edge in the graph G. If $p \simeq q$ holds, so does $p \circ m \simeq q \circ m$.
- Let $p, q : a \to b$ be paths and $n : b \to c$ be an edge in the graph G. If $p \simeq q$ holds, so does $n \circ p \simeq n \circ q$.

Finally, the schema of a database in categorical normal form is a pair $(G, \simeq)$, where $G = (V, E, src, tgt)$ freely generates the category $\mathbf{G} = (\mathcal{O}_\mathbf{G}, \mathcal{M}_\mathbf{G}, \circ, 1)$ as follows:

- $\mathcal{O}_\mathbf{G}$ correspond to the vertices in V.

[19] https://www.categoricaldata.net

- $\mathcal{M}_{\mathbf{G}}$ correspond to the paths in the graph G composed of edges E.
- $\circ$ corresponds to the path concatenation operation.
- $\simeq$ is a categorical path equivalence relation on G.

In other words, the kinds (i.e., tables) in the schema are determined by vertices $v \in V$ mapped to objects $O \in \mathcal{O}_{\mathbf{G}}$, the properties (i.e., columns) are determined by edges $e \in E$ mapped to morphisms $m \in \mathcal{M}_{\mathbf{G}}$, and the integrity constraints are represented by a categorical path equivalence relation. A closer look allows us to further divide the objects into (1) objects $T \in \mathcal{O}_{\mathbf{G}}$, each corresponding to a kind of a single property (i.e., a single one-column relational table) and (2) objects $D \in \mathcal{O}_{\mathbf{G}}$, each corresponding to a generic data type, e.g. String, Number, Boolean, etc. As for morphisms, we can distinguish: (1) identity morphisms $id_t : T \to T$, each modelling an identifier of a specific kind (i.e., relational table); in the case where the kind has no identifier, id_t is considered implicitly, (2) identity morphisms $id_d : D \to D$, each modelling an identifier for a domain of a particular data type, (3) morphisms $r : T \to T'$, each modelling a reference from T to T', and (4) morphisms $c : T \to D$, for each kind T and its property (distinct from the identifier and reference) of data type D. Also note that the categorical normal form does not support complex and overlapping identifiers.

The authors also introduce a *set-valued functor* $Inst_{\mathbf{G}} : \mathbf{G} \to \mathbf{Set}$ representing a particular instance conforming to the schema $\mathbf{G}$ (see Example A.5).

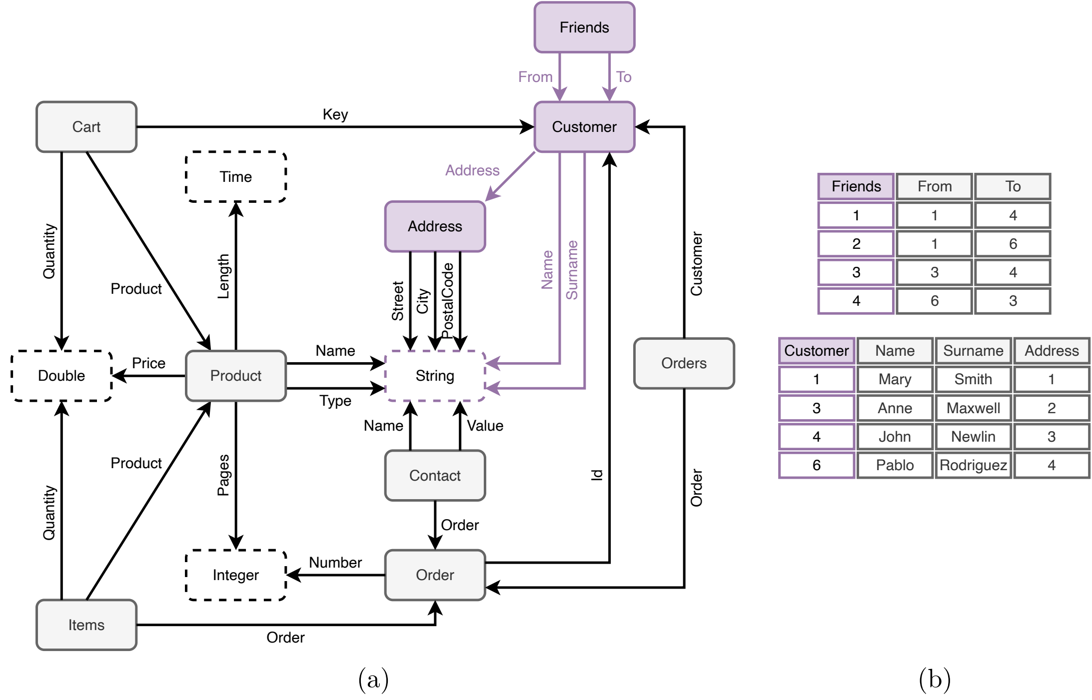
Example 0.7. Figure 0.6 (a) illustrates the multi-model schema from Figure 0.1 represented by the approach [20]. Objects depicted using a solid line represent particular tables (e.g., Customer), while objects depicted with a dashed line represent data types (e.g., String). Although the schema appears to faithfully capture the logical schema of multi-model data, it is in fact expressed in terms of interconnected single-column (non-aggregated) relational tables. Moreover, this approach does not consider complex or overlapping identifiers, but only trivial one-column identifiers. Also note that instead of an ISA hierarchy, the specialisation of a child is represented by addition of the property $Type$ to the kind $Product$, as the approach [20] does not allow explicit modelling of the ISA hierarchy.

Finally, the part of the instance is illustrated in Figure 0.6 (b). □

Papers [56, 57] extend this approach to support multiple logical models, i.e., not just the relational model. However, the proposal only considers separate models (namely relational, graph, and document) over which the data can be queried and migrated between using functors and natural transformations.

Algebraic Property Graph (APG)

Yet another approach [21] is based on type theory, algebra, and category theory, i.e., it is closely related to algebraic databases [58], and aims to cover the area of property graphs. The authors propose the so-called *algebraic property graph* (APG) to represent a general property graph. In addition, a set of rules for transforming selected data models (i.e., relational, RDF, key/value, XML and, JSON document) into the APG representation is proposed. Also, the basic operations of



Friends	From	To
1	1	4
2	1	6
3	3	4
4	6	3

Customer	Name	Surname	Address
1	Mary	Smith	1
3	Anne	Maxwell	2
4	John	Newlin	3
6	Pablo	Rodriguez	4

Figure 0.6: An example of a logical categorical schema and data (Spivak et al.)

querying and migration between two APGs are proposed. The whole framework is implemented as a part of an open-source tool CQL.[20]

Property graph $G = (V, E, L, src, tgt, lbl_v, lbl_e)$ is defined as follows:

- V is the set of vertices of the graph G.
- E is the set of edges of the graph G.
- Functions src, tgt realize the edge orientation.
- Unlike the ordinary graph, G also contains a set of labels L and functions $lbl_v : V \to L$, $lbl_e : E \to L$ that assign labels $l \in L$ to vertices $v \in V$ and edges $e \in E$ accordingly.

Note that internally each vertex and edge contains a unique identifier and a set of key/value pairs called properties.

The authors represent the property graph as a category $\mathbf{A} = (\mathcal{O}_\mathbf{A}, \mathcal{M}_\mathbf{A}, \circ, 1)$, where $\mathcal{O}_\mathbf{A} = \{L_\mathbf{A}, T_\mathbf{A}, E_\mathbf{A}, V_\mathbf{A}\}$ and $\mathcal{M}_\mathbf{A} = \{\upsilon_\mathbf{A}, \tau_\mathbf{A}, \lambda_\mathbf{A}, \omega_\mathbf{A}\}$ such that:

- $L_\mathbf{A}$ represents the set of labels to be assigned to each vertex of the graph.
- $T_\mathbf{A}$ denotes the set of types that can be assigned to the vertices of the graph. Each type $t \in T_\mathbf{A}$ is a term of the grammar $t := 1 \mid p \mid l \mid t_1 + t_2 \mid t_1 \times t_2$, where 1 is a type of a trivial value, $p \in P$ is a primitive type, $l \in L_\mathbf{A}$, $t_1 + t_2$ is a complex type (e.g., union type), and $t_1 \times t_2$ is a type of an edge.
- The set of elements $E_\mathbf{A} = V \cup E$ represents the elements of the graph G, i.e., the vertices and edges.

[20] https://www.categoricaldata.net

- The set $V_{\mathbf{A}}$, where the elements $v_i : t_i$ are terms of the typed grammar $v : t := () : 1 \mid inl_{t_2}(v : t_1) : t_1 + t_2 \mid inr_{t_1}(v : t_2) : t_1 + t_2 \mid (v_1, v_2) : t_1 \times t_2 \mid v_p : p \mid e : \lambda_{\mathbf{A}}(e)$, where $t, t_1, t_2 \in T_{\mathbf{A}}$, $p \in T_{\mathbf{A}}$ is a primitive type, and $e \in E_{\mathbf{A}}$.

- The function $v_{\mathbf{A}} : E_{\mathbf{A}} \to V_{\mathbf{A}}$ assigns a value to each element.

- The function $\tau_{\mathbf{A}} : V_{\mathbf{A}} \to T_{\mathbf{A}}$ associates a data type with each value.

- The function $\lambda_{\mathbf{A}} : E_{\mathbf{A}} \to L_{\mathbf{A}}$ attaches a label to each element.

- The function $\omega_{\mathbf{A}} : L_{\mathbf{A}} \to T_{\mathbf{A}}$ determines the (data) type of each label.

- The structure of the graph must correspond to its schema, i.e., $\tau_{\mathbf{A}} \circ v_{\mathbf{A}} = \omega_{\mathbf{A}} \circ \lambda_{\mathbf{A}}$.

A particular algebraic property graph (i.e., an instance) is then a functor $F : \mathbf{A} \to \mathbf{Set}$ (see Example A.5).

Example 0.8. An example of $l \in L_{\mathbf{A}}$ is, e.g., `Customer` attached to a vertex representing a particular customer, and `name` attached to an edge that assigns a name to a customer.

Examples of $t \in T_{\mathbf{A}}$ include `Customer` (i.e., the type of vertex attached by label `Customer`), `String` (i.e., the type of a vertex that represents an attribute), `Customer` $\times$ `String` (i.e., the type of an edge that assigns a name of type String to the Customer), and `Product + Audiobook` (i.e., the type of vertex that has multiple labels attached).

An example of a typed value is, e.g., `"Mary":String`, `():1` (i.e., a trivial value of an arbitrary vertex), `(`c_1`, "Mary"):Customer` $\times$ `String` (i.e., an edge value, where c_1 is a reference to the vertex corresponding to the customer with $id = 1$).

Then the function $\lambda_{\mathbf{A}}(c_1) = Customer$ assigns the label Customer to the vertex c_1. The function $v_{\mathbf{A}}(n_1) = (c_1, "Mary")$ determines the value of the edge referenced by the reference n_1, or $v_{\mathbf{A}}(c_1) = ()$. Examples of exploitation of the function to determine the label of a graph element are $\omega_{\mathbf{A}}(Customer) = Customer$, and $\omega_{\mathbf{A}}(name) = Customer \times String$. Finally, an example of determining the type of a typed value is $\tau_{\mathbf{A}}("Mary", String) = String$. □

Example 0.9. Figure 0.7 (a) illustrates the schema of the data from Figure 0.1 represented as APG. The green objects correspond to real-world objects, the black and white objects represent property values, and the blue arrows, each crossing a blue object, represent directed relationships between the objects. Note that properties of objects or relationships are represented by the edge leading to the black and white objects.

Figure 0.7 (b) illustrates a part of the multi-model data from Figure 0.1 that corresponds to the schema in Figure 0.7 (a). Note that the approach allows us to represent the ISA hierarchy by attaching multiple labels to a single vertex. Moreover, vertices are identified only by a simple implicit identifier, i.e., complex identifiers are not allowed. Finally, the structured attribute Address is inlined to the Customer. □

APG is also suitable for data and schema migration. As every instance of APG corresponds to a functor $F_{1,2} : \mathbf{A} \to \mathbf{Set}$, the migration between two instances of

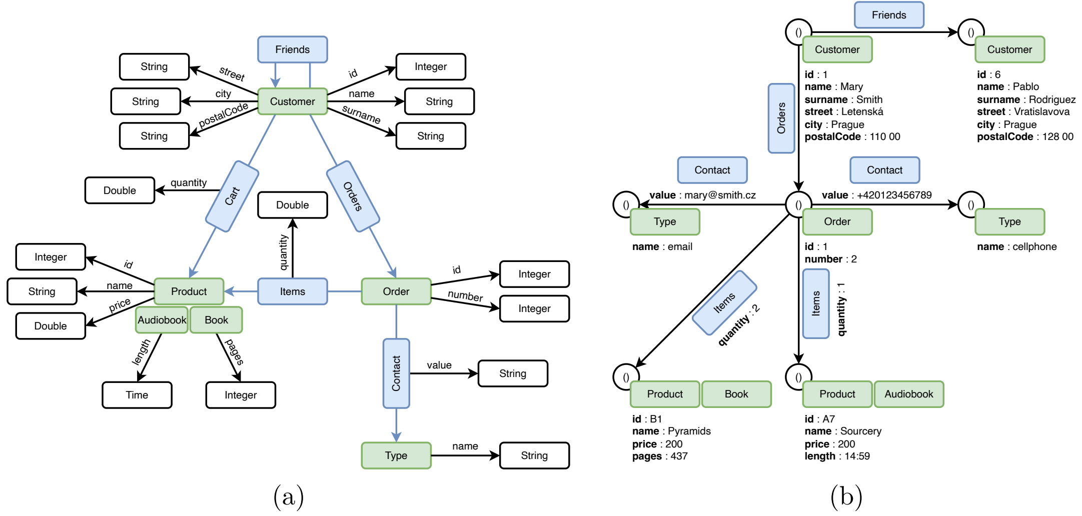


Figure 0.7: An example of APG (a) corresponding to data (b)

APGs can be formally described as a natural transformation $\alpha : F_1 \to F_2$ (see Definition 10).

Finally, the authors also propose a method for APG to represent the schema of other data models (i.e., relational, RDF, key/value, XML and JSON document), but do not cover all the properties of the models, e.g., uniqueness of values, order of properties, structured and overlapping identifiers.

NoSQL Abstract Model

The approach [27] is a system-independent model of aggregate-oriented NoSQL database systems, i.e., it covers key/value, document, and columnar models. In addition, it is designed for performance, scalability and data consistency.

The basic constructs of the proposed abstract model are as follows:

- The NoAM database $D = \{C_1, \ldots, C_n\}$ is a set of collections.
- Each collection $C = (id_C, B)$ is uniquely identified by the collection key id_C and contains the set of blocks $B = \{b_1, \ldots, b_m\}$. Examples of collections include bucket, document collection, and column family.
- Each block $b = (id_b, E)$ is uniquely identified by the block key id_b and contains a non empty set of entries $E = \{e_1, \ldots, e_p\}$. Each block corresponds to an aggregate, e.g., a single key/value pair, a document, or a row of a column family. A block is also the largest data unit for which atomic operations are considered.
- Entry $e = (e_k, e_v)$, where e_k uniquely identifies an entry within a block and e_v represents a primitive or complex value. An entry corresponds to a document field and a table column.

NoAM employs two data representation strategies, i.e., *input per each top-level property* (ETF) and *input per each aggregated object* (EAO).

When applying the ETF strategy, each block b represents a single record, wherein the block key id_b corresponds to the identifier of the record, and the set

of entries E contains one pair $(e_k,\ e_v)$ for each property except for the property having the identification feature. If the top-level property represents a complex property, i.e., it is not an atomic property, this property is also represented as an entry and not as a set of nested entries (e.g., a nested block).

Alternatively, when applying the EAO strategy, each record is represented by its own block, where the block key id_b corresponds to an identifier and the set of entries is constituted by a single entry $(e_k,\ e_v)$. Note that $e_k = \epsilon$ and the value is the aggregate corresponding to the record without the identifier (as it is already used as id_b).

Example 0.10. Figure 0.8 (a) illustrates the application of the ETF strategy to the kinds *Order* and *Customer* from Figure 0.1, while Figure 0.8 (b) illustrates the application of the EAO strategy to the same data. A comparison of the two strategies demonstrates that the block identifiers are the same and the blocks differ only internally. While the block contains only a single complex entry identified by the (meta)value ϵ when using the ETF strategy, the block consists of a set of uniquely recognisable entries when using the EAO strategy. However, even if using the EAO strategy, the entries corresponding to a complex structure are not split further into smaller units (e.g., see properties contact and items). □

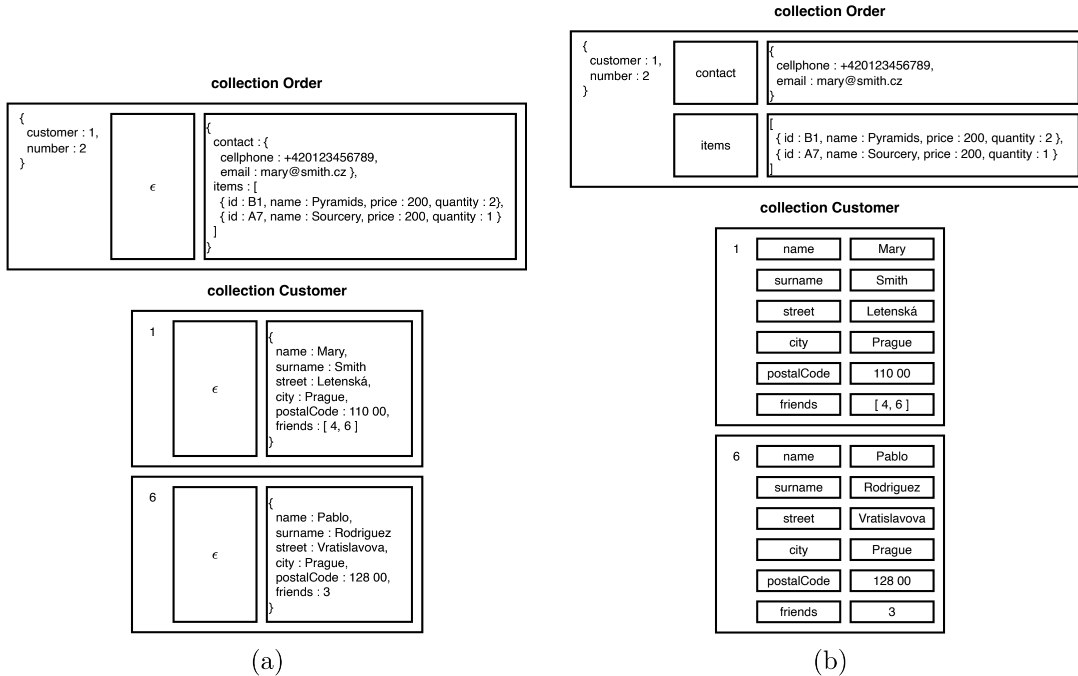


Figure 0.8: An example of (a) ETF and (b) EAO strategies

In addition, the so-called access paths ap are considered, allowing to represent the complex property p as a set of entries $\{(ap_1, v_1), \ldots, (ap_n, v_n)\}$. As such, each nested property c is represented as an entry (ap_c, c), where the entry key ap_c is the sequence of steps required to access property c from parent p, and the value c is the accessed nested property.

In order to select an appropriate aggregate representation strategy, the authors propose a set of rules for partitioning the data model. As the authors suggest, the chosen strategy should reflect, e.g., data access patterns and support

strong consistency and efficient execution of update operations. Unfortunately, the approach does not consider general multi-model data, i.e., a combination of possibly overlapping data models, but only aggregate-oriented data. Note also that the approach only allows for so-called embedding of kinds (i.e., nested complex properties representing an additional structure), but it does not consider references between different kinds (i.e., across different data collections) at all.

Associative Arrays

The next approach [28] is a system-independent model that allows selected data models to be represented uniformly at the logical and physical layers. As such, it is based on a generic data structure, so-called *associative arrays*.

The core of the approach is the associative array A, i.e., a mapping of a two-dimensional key to the value $A : K_1 \times K_2 \rightarrow \mathbf{V}$, where:

- Key K_1 corresponds to an array row.
- Key K_2 corresponds to an array column.
- A domain of a key can be an arbitrary set of ordered values, e.g., integers, strings, etc.
- The values $k \in K_i$ must be unique within the key K_i, i.e. there does not exist $k_i, k_j \in K : k_i = k_j$, but there may exist two keys $k_1 \in K_1$, $k_2 \in K_2$ such that $k_1 = k_2$.
- There is no row or column that is completely empty, i.e., an associative array does not allow us to represent an empty record or a property having an empty active domain.

Associative arrays allow to represent both aggregate-ignorant and aggregate-oriented models. In the former case, the models are represented as a matrix where the row key corresponds to the recorder identifier and the column key corresponds to the property name, while in the latter case it is a sparse matrix that additionally represents a hierarchical arrangement of data in the keys. For example, the row key contains not only the document identifier, but additionally a unique index for nested array elements to determine their order, and the column key reflects the hierarchy between properties, i.e., top-level properties are represented by the property name, while further nested properties are represented by the minimum sequence of steps required to access the property (similarly to the access paths in NoAM). However, the approach does not model, e.g., references, multiple (possible overlapping) identifiers, and the ISA hierarchy at all.

Example 0.11. Figure 0.9 illustrates the application of the approach [28] to the multi-model data from Figure 0.1. As illustrated, the approach allows to represent heterogeneous models as a disjunctive set of associative fields. Note that the approach allows to represent a complex identifier or complex and nested properties, including capturing the order of elements in the array, as illustrated by associative array representing kind *Order*. □

The proposed approach also provides querying and transformations. It uses basic operations, e.g., element-wise addition, element-wise multiplication, and

	name	surname	street	city	postalCode
1	Mary	Smith	Letenská	Prague	110 00
3	Anne	Maxwell	Ke Karlovu	Prague	110 00
4	John	Newlin	Technická	Prague	162 00
6	Pablo	Rodriguez	Vratislavova	Prague	128 00

	name	surname	1	3	4	6
1	Mary	Smith	0	0	1	1
3	Anne	Maxwell	0	0	1	0
4	John	Newlin	0	0	0	0
6	Pablo	Rodriguez	0	1	0	0

	contact/ cellphone	contact/ email	items/ id	items/ name	items/ price	items/ quantity
(customer : 1, number : 2)	+420123456789	mary@smith.cz				
(customer : 1, number : 2)/001			B1	Pyramids	200	2
(customer : 1, number : 2)/002			A7	Sourcery	200	1

	cart
1	product : T1, quantity : 2, product : B4, quantity : 1
3	product : H1, quantity : 1
6	product : B3, quantity : 2

	01	02	03	...
1	1,1	1,2	...	...
3	3,1	3,2		
4	4,1	4,2	4,3	

Figure 0.9: An example of Associative Arrays

array multiplication, which correspond, e.g., to the database operations of table union, intersection, and transformation.

Tensor Data Model (TDM)

Last but not least, the approach [29] allows us to represent multi-model data in terms of tensors [59], i.e., the following matrix generalisation: 0th order tensor is a scalar, 1st order tensor is a vector, 2nd order tensor is a matrix, and n-th order tensor is so-called *higher-order tensor*.

In TDM, tensors are defined as the mapping $T : K_1 \times \cdots \times K_n \to \mathbb{V}$, where: $n \in \mathbb{N}$ is the order of the tensor, K_i is the dimension of the identifier $K_1 \times \cdots \times K_n$, and $\mathbb{V}$ is a set of values. In addition, tensors are unambiguously named and typed.

Tensors are used in three possible ways in TDM:

1. Associative arrays, denoted by A_i for $i = 1, \ldots, n$, model dimensions of a tensor X. Such arrays have only one set of keys associated with integers using bijective function $A_i : K_i \to \mathbb{N}$.

2. At a lower level, an associative array is used to represent the values of a sparse n-order tensor by associating compound keys from dimensions to values $A_{vst} : K_1 \times ... \times K_n \to \mathbb{V}$.

3. For tensors with non-numerical values, two associative arrays are used: (i) to map keys dimensions to a set of integer keys (A_{vst}) and (ii) to map the integer keys to non-numeric domains values (one integer is associated with each different value).

Operations with tensors are analogous to operations with matrices and vectors, e.g. multiplication, transpose, unfolding (transformation of a tensor into a matrix), factorisation (decomposition), etc.

Example 0.12. Figure 0.10 illustrates the application of the approach [29] to the multi-model data from Figure 0.1. At first sight, the approach is identical to associative arrays proposed in [28]; however, the document and column models are

not considered. That is, a hierarchy between individual properties and complex properties such as nested documents, arrays, etc. cannot be represented. On the other hand, this approach allows us to apply higher-order tensors, e.g., to represent n-ary relations. In this case, we could use a third-order tensor to represent, e.g., graph data illustrating relationships between two customers. □

	name	surname	street	city	postalCode
1	Mary	Smith	Letenská	Prague	110 00
3	Anne	Maxwell	Ke Karlovu	Prague	110 00
4	John	Newlin	Technická	Prague	162 00
6	Pablo	Rodriguez	Vratislavova	Prague	128 00

	name	surname	1	3	4	6
1	Mary	Smith	0	0	1	1
3	Anne	Maxwell	0	0	1	0
4	John	Newlin	0	0	0	0
6	Pablo	Rodriguez	0	1	0	0

	cart
1	product : T1, quantity : 2, product : B4, quantity : 1
3	product : H1, quantity : 1
6	product : B3, quantity : 2

Figure 0.10: An example of TDM

U-Schema

Recently, the approach [30] integrating schemas of distinct logical models, namely relational, graph, key/value, document and columnar models, was proposed. The proposal also includes a mapping between the respective logical model schema and the integrating schema and vice versa.

The basic structural constructs of the proposed approach are as follows:

- Model U-Schema $U = \{s_1, \ldots, s_n\} \subseteq S$, $n \in \mathbb{N}$ is a set of SchemaTypes $s \in S$.

- The set of SchemaTypes S is the union of EntityTypes ($E \subseteq S$) representing classes of real-world objects and RelationshipTypes ($R \subseteq S$) representing relationships between them. Internally, EntityType $e \in E$ is represented as a tuple $(n_s, root, V_s)$ and RelationshipType $r \in R$ is a tuple (n_s, V_s), i.e., each SchemaType is assigned a name n_s and contains a subset of Structural-Variations $V_s \subseteq V$. In addition, EntityType e contains a Boolean feature $root$ indicating whether it is a standalone or nested entity type. Hence, only the EntityType can form a hierarchical structure.

- Each StructuralVariation $v \in V$ is represented as a tuple (id, F_v, $count$, $firstTS$, $lastTS$) such that id is an integer identifier, $F_v \subseteq F$ is a subset of the properties, $count$ is a feature capturing the number of existing instances of a particular StructuralVariation, and the timestamps $firstTS$ and $lastTS$ store the time of creation of the first and last instance of that variation.

- The set of properties F is composed of LogicalFeatures (i.e., keys $KEY \subseteq F$ and references $REF \subseteq F$) and StructuralFeatures (i.e., attributes $ATT \subseteq F$, and aggregates $AGG \subseteq F$) such that:

- The attribute $att \in ATT$ can be primitive (e.g., Number, String, Boolean, JSON,[21] BLOB), collections (e.g., list, tuple, set, map), or the special type Null. Each attribute att is modeled as a tuple (n_{att}, t_{att}, opt_{att}, key_{att}, REF_{att}, is_{att}) such that n_{att} assigns a name to the attribute, t_{att} is the data type of the attribute, opt_{att} specifies the optionality of the attribute, $key_{att} \in KEY$ is a reference to the (none or only) key of which the attribute may be a part, $REF_{att} \subseteq REF$ is a subset of the references of which the attribute is a part, and the is_{att} attribute models specific behaviour based on the attribute type.
- The key $key \in KEY$ is modelled as a tuple (n_{key}, A_{key}) such that n_{key} is the name of the key and $A_{key} \subseteq ATT$ is a subset of attributes that constitute the key.[22]
- The reference $ref \in REF$ is modelled as a tuple (n_{ref}, A_{ref}, $refsTo$, $lBound$, $uBound$) such that n_{ref} is the name of the reference, $A_{ref} \subseteq ATT$ is a subset of the referencing attributes,[23] $refsTo \in E$ is the referenced EntityType, and $lBound$, $rBound$ are the lower and upper cardinality bounds, respectively.
- The aggregation $agg \in AGG$ is modeled as a tuple (n_{agg}, opt_{agg}, $lBound$, $uBound$, V_{agg}) such that n_{agg} assigns a name to the aggregation, opt_{agg} determines the optionality of the aggregation, $lBound$, $rBound$ are the lower and upper cardinality bounds, and $V_{agg} \subseteq V$ is the set of structural variations that are aggregated (nested).

Example 0.13. Figure 0.11 (a) illustrates the U-Schema of the graph data and Figure 0.11 (b) the U-Schema of the document data from the Example 0.1. Note that although edges are represented internally in the graph as a pair of properties from, to, U-Schema represents an edge as a RelationshipType without any properties, and the connection between two customers (otherwise realised by an edge) is represented by a reference at Customer. In the case of the U-Schema document model, note that only the EntityType Order is root, reflecting the fact that all other EntityTypes are nested. □

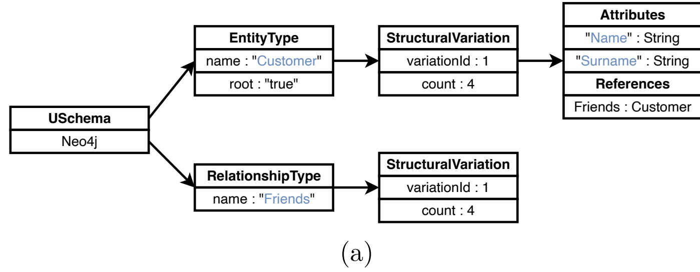
U-Schema allows to define two variants of the model: (1) in the so-called *full variability* all structural variations of all EntityTypes and RelationshipTypes are stored, while (2) in the so-called *union schema* there is only one structural variation for each SchemaType. Note that the conversion from the *full variability* to the *union schema* version is a lossy conversion, hence the reverse conversion is not possible.

Furthermore, the authors introduce so-called forwards mapping (i.e., a mapping of the logical model schema to the U-Schema) and reverse mapping (i.e., a mapping in the opposite direction). In the former case, there is a natural correspondence between each element of the logical schema and an element of the U-Schema. However, in the latter case, the U-Schema may contain elements that

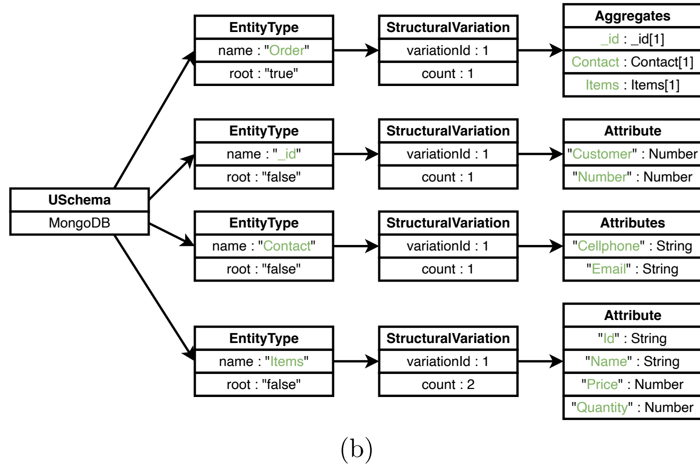
[21] Note that the authors consider the nested data model represented by JSON or JSONB type in PostgreSQL as a black box and not as a structure.

[22] Note that U-Schema allows primitive attributes as part of the key, as well as collections and the special type Null [30].

[23] Note that, similar to the key, references can be collections and the special value Null in U-Schema.



(a)



(b)

Figure 0.11: An example of U-Schema for graph (a) and document model (b)

are not present in a particular logical model, e.g., the relational model does not contain structural variations,[24] the graph model does not support aggregates, and conversely, most logical models do not contain the RelationshipType. Hence, the unification of data models is limited, as model-specific constructs are introduced into U-Schema, which also makes it difficult to extend the approach to support other logical models if needed.

Finally, the authors proposed the language Athena [60], which allows the definition of logical schemas as U-Schemas.

Comparative Summary

Table 0.3 summarises the features of selected abstract models at the logical layer. All the observed approaches allow us to represent data, however, the schema is approached in various ways. Most approaches can be considered schema-full, however, NoAM approach lacks an explicit schema, thus representing a schema-less approach. The abstraction of logical data models varies. Typically, if the abstract model is based on the graph (or category), it allows to represent relational or graph data (i.e., aggregate-ignorant models). Conversely, if the model is an array-like or aggregate-like, mostly aggregate-oriented models are supported. However, multi-model data is only considered in the U-Schema approach, and

[24]Note that the U-Schema treats missing value differently. While in the relational model, missing value leads to an optional property within one structural variation, in the graph model, missing value leads to two different structural variations, and in the document model we further distinguish a variation with the special type Null.

Table 0.3: Comparison of logical layer modeling approaches

	CGOOD [19]	Spivak et al. [20]	APG [21]	NoAM [27]	AA [28]	TDM [29]	U-Schema [30]	MM-cat [3]
Data structure	Typegraph	Category	Category	Aggregate	Matrix	Tensor	Graph	Category
Schema	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Data	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
Relational model	Partial	Partial	Partial	No	Partial	Partial	Partial	Yes
Array model	No	No	No	No	No	No	No	Yes
Graph model	No	No	Yes	No	Yes	Yes	Yes	Yes
RDF model	No	Yes	No	No	No	No	No	Yes
Key/value model	No	No	No	Yes	Yes	Yes	Yes	Yes
Document model	No	No	No	Yes	Yes	No	Yes	Yes
Columnar model	No	No	No	Yes	Yes	No	Yes	Yes
Multi-model data	No	No	No	No	No	No	No	Yes
Complex ID	No	No	No	Yes	Yes	No	Yes	Yes
Multiple IDs	No	No	No	No	No	No	No	Overlapping
References	Implicit	Intra	Implicit	No	No	No	Yes	Intra, Inter
Complex IC	No	∼	∼	No	No	No	No	No
Complex types	Structure, Array	No	Array	Structure, Array	Implicit	No	Structure, Array, Tuple, Set, Map	Structure, Array, Tuple, Set, Map
Union type	No	No	Yes	Implicit	No	No	Yes	Yes
Ordering	No	No	No	Yes	Yes	No	No	Yes
Data redundancy	No	No	No	No	No	No	No	Yes
Querying	Graph patterns	CQL	CQL	No	Matrix algebra	Tensor algebra	No	No
Data Migration	Intra	Yes	Yes	No	No	Yes	Yes	Yes
Evolution management	Yes	Partial	Partial	No	No	Partial	Yes	Yes

only in a limited way. The approach treats multi-model data as a set of disjunctive models, i.e., it does not consider features arising from their connections, e.g., inter-model references, redundancy (see Chapter 3) and (in)consistency [61]. ★

Nevertheless, even in the case of support for individual data models, approaches are often limited to a minimal set of constructs, e.g., simple identifiers, simple properties and basic data structures (i.e., a subset of complex properties). Only approaches NoAM and AA allow to model complex identifiers, while completely missing, e.g., the possibility to specify multiple identifiers and thus to represent overlapping identifiers typical, e.g., for a relational model, hence multi-model data. In the case of references, the approaches are limited to (implicit) intra-model references only.

Finally, approaches often introduce minimal sets of operations that can be composed to perform more complex operations that express, e.g., querying, data migration and some evolution management operations.

0.3.3 Open Questions and Challenges in Data Modelling

Indeed, existing data modelling approaches seem promising for representation of multi-model data at the conceptual and logical layers. However, none of these approaches allows us to represent all the features of multi-model and underlying data in their natural form. Hence, there remain open questions that need to be addressed.

Conceptual Layer

Naturally, approaches that model the conceptual layer should support multi-model data, as they intentionally hide the specific properties of platform-specific (logical) representations. However, traditional approaches, such as ER and UML, are primarily closely associated with the relational model (i.e., normalised data) and do not necessarily reflect the properties of additional models (e.g., denormalised data). In other words, the translation of the conceptual schema is not straightforward and is often ambiguous in the case of a combination of models.

The following **challenges C1 − C9** need to be discussed accordingly in order to design an extended approach that fully represents multi-model data at the conceptual level:

C1: *Elements of the conceptual layer.* The question is whether we need to distinguish classes of objects from relationships and properties at the conceptual level. In practice, conceptually equivalent constructs are often represented differently at the logical level, and different conceptual constructs are represented in the same way. This is particularly obvious in aggregate-oriented models where, e.g., a structured attribute is interchangeable with a combination of a class of parent object, a class of (nested) objects, and respective relationships.

C2: *Updated concept of a property.* The traditional concept of a property can be understood as a pair $(name, value)$, where *name* is statically bound to *value* (thus easily representable at the conceptual level). However, with the advent of data models that allow data to be represented by a property

of type map, the static binding is replaced by a dynamic one, i.e. *name* becomes a part of the data. This trait, i.e., the dynamic naming of a (nested) property, would also be useful to represent at the conceptual layer.

C3: *Required explicit/implicit identifier.* We believe that the motivation for the introduction of a required and explicit identifier for classes of objects at the conceptual level was, among other things, to enable the unambiguous connection of different instances of classes, i.e., to realise relationships between them, whereby a particular instance of a relationship is implicitly identified just by the participants. Moreover, properties can be seen as trivial object classes that contain only an identifier and thus explicitly identify themselves [20]. Hence, it is possible to identify all the elements of conceptual model, which could allow a further level of unification.

C4: *Ordering of properties.* Currently, the conceptual level only allows to represent property names within a single class, but their relative order cannot be captured.

C5: *Structured property.* Arguably, given the properties of the relational model, a structured property, e.g., in ER, can only have trivial depth and allows only trivial cardinality (i.e. one-to-one). However, in practice we also encounter unrestricted structured properties that have non-trivial depth and where sub-properties are repeated. Traditionally, such a state can be represented by a combination of object classes and relations, but this goes against the concept of conceptual modelling, as it is a structured property and not a combination of classes and relationships.

C6: *Multivalued identifier.* Considering simple (single-property) identifiers, currently only properties with trivial cardinality (i.e., one-to-one) can be used to identify an instance within an object class.

C7: *Identifier of a weak class type.* The identifier of the weak class type is formed by the identifiers of all types involved in the relationship. In some cases, it may be sufficient for the weak identifier to consist of only a subset of the identifiers of the types participating in the relationship.

C8: *Complex integrity constraints.* Ideally, we need to capture integrity constraints (even complex ones) at the conceptual layer, as integrity constraints are a natural and often overlooked part of the schema. In particular, ER is limited to the representation of identifiers, whereas UML using OCL allows the description of multiple types of integrity constraints including, e.g., business rules.

C9: *Aliasing properties.* Each element of the conceptual schema is assigned only with a single name. However, in certain cases it is useful to introduce synonyms, e.g., properties `father` and `mother` could be referred to as property `parent`. Consequently, the special case of inheritance, though at the property level, could, e.g., simplify (conceptual) querying.

Logical Layer

Based on our findings, a fundamental construct of popular logical data models is a property, i.e., the mapping $p : N \to V$, where $n \in N$ is the name and $v \in V$ is the value. Yet the models differ, e.g., in the way the properties are aggregated into larger logical units (e.g., a flat table or a hierarchical tree structure), in the enforcement of the order of properties (i.e., order-oriented/ignorant models), the possibilities of identifying larger units, the support for references between larger units, the way missing data is represented, or the attachment of structural information to the data (i.e., structured, semi-structured, unstructured formats). A common underlying construct (i.e., name-value pair) can be used to design a unified (abstract, not necessarily logical) model. In designing such a model, the following **challenges L1 − L7** must be properly tackled:

L1: *Unified logical model.* A question is whether we need a unified layer of multi-model data at the logical level in the sense of a single data model (more or less painfully combining the properties of the models) or rather an abstract model that merely overlays the existing logical models. In the latter case, we could continue to take advantage of features of existing approaches, e.g., representing data as a graph if we query primarily over relationships, or aggregates if we repeatedly call queries aggregating data, etc., while treating all models uniformly.

L2: *Representation of model-specific constructs.* The unifying layer should allow for uniform capturing of semantically similar constructs across different models (e.g., a nested JSON document as a map or a tuple as an array) and also capture model-specific constructs (e.g., complex and overlapping identifiers).

L3: *Missing data.* Logical models differ in the way they represent missing data. For instance, the relational model represents missing data as a null (meta)value, the graph model (e.g., property labelled graph implemented in Neo4j) as non-existent properties, while the document model allows combining both approaches. The question is whether it is possible to represent missing data in a unified abstract way.

L4: *Inter-model links and references.* Logical models support various forms of intra-model references (e.g., traditional references and embedding). However, references and embedding across different models are a natural feature of multi-model data. For example, PostgreSQL allows embedding of XML and JSON document data into a relational table by introducing a special column type.[25] Hence, we need an abstract model that considers both intra- and inter-model references.

L5: *Conceptual to logical layer mapping.* Ideally, we need a unified algorithm that allows mapping between conceptual and logical layers, regardless of the specific properties of the logical layer.

[25] https://www.postgresql.org/docs/current/

L6: *Propagation of changes.* The unified data model should be designed with a perspective of a uniform way of propagating changes in different (underlying) logical models, but also across these models. The propagation of schema and data changes should also be accompanied with the propagation of query changes.

L7: *Extensibility* towards new data models that currently does not exists but are based on similar idea, e.g., where the most general construct is a pair $(name, value)$.

0.3.4 Contribution: Framework MM-cat

So far, we have discussed existing approaches and challenges that need to be addressed in order to appropriately represent multi-model data at the conceptual and logical level. Now, we turn to a commentary of the actual solution and its incremental extension to meet additional requirements, i.e., not just representing multi-model data in a unified way, but enabling data migration, querying, and evolution management. In this subsection, we mainly discuss data representation, while migration and evolution management are addressed in separate Section 0.5 and querying forms our future work.

We have first analysed selected existing data representation solutions at the conceptual level [31, 32, 18] and logical level [19, 20, 21, 27, 28, 29], as well as other approaches [48, 62] describing, e.g., integrity constraints, and we have verified their applicability to multi-model data [26], thereby also revealing drawbacks and open questions. Being aware of the limitations of the solutions, we have ★ outlined the concept of a unified schema representation [63] and the vision of a comprehensive framework built on top of solid formal foundations and allowing ★ the management of multi-model data in a unified yet natural way (see Chapter 1).

As mentioned above, the development of our approach has been gradual. In the beginning, we mainly considered schema and data representation at the conceptual level. The goal of developing an early concept was to test whether we could apply category theory at all to the representation of multi-model data. In the second stage, we added support for logical-level mappings and introduced the notion of mapping between logical and conceptual levels. In addition, we used this mapping in the design of algorithms implementing data migration. Finally, we have extended the approach to naturally support complex schema and data change operations, i.e., to enable evolution management.

Early / Original Concept

In the early stages of designing the unifying conceptual model, we tried to represent multi-model data as simply as possible, i.e., using only the elementary constructs of category theory (e.g., categories, functors, and their composition). The motivation for this decision was the desire to create a model that has the potential for broad extension and application, and thus it is appropriate to avoid complex constructs at this elementary level. Our inspiration came from the works [18, 19, 20] modelling conceptual, object-relational, or relational schemas, as well as vision [56], in which the authors outlined an extension of the [20] approach towards supporting additional data models.

Our approach (see Chapter 2) is based on definition of a schema category, an instance category, and their mutual mapping. The *schema category* $\mathbf{S} = (\mathcal{O}_\mathbf{S}, \mathcal{M}_\mathbf{S}, \circ_\mathbf{S}, 1)$ captures the data structure and describes the basic integrity constraints. Objects in $\mathcal{O}_\mathbf{S}$ represent the classes of real-world objects, relationships between them, and properties of both object and relationships (challenge C1) in a uniform way. Although the user may still distinguish between the three object types (i.e., object, relationship, and property), this is not necessary from the categorical point of view. Additionally, an object is internally represented as a pair (*superid*, *ids*), with *superid* representing a concept of a superidentifier and *ids* being a set of identifiers. Such representation allows us explicitly include the identifier also for relationships and properties, if appropriate (challenge C3). Morphisms in $\mathcal{M}_\mathbf{S}$ model the concept of object relations. Internally, a morphism is represented as a pair (min, max), which allows us to model the traditional concept of cardinalities, while at this stage we only consider trivial cardinalities having $min \in \{0, 1\}$, $max \in \{1, *\}$, $min \leq max$. The composition of morphisms $\circ_\mathbf{S}$ also reflects the composition of cardinalities. For convenience, the user can also still distinguish morphisms based on the type of linked objects into property, relation, and ISA hierarchy morphisms, but from a categorical point of view this is again not essential. ★

Instance category $\mathbf{I} = (\mathcal{O}_\mathbf{I}, \mathcal{M}_\mathbf{I}, \circ_\mathbf{I}, 1)$ represents data in a unified way. This category structurally corresponds to the schema category, i.e., there is a functor $F : \mathbf{I} \to \mathbf{S}$ that assigns a schema to the data (as inspired by the approach [19]). Alternatively, we can represent the instance category as a category of sets and functions, i.e., we can build it on top of category **Set** (this variant is described in Chapter 2) or **PartSet** allowing to additionally represent missing data (as inspired by the approach [20]). In contrast, if we were to represent schema and data directly using graph theory or set theory instead of category theory, the choice of instance representation would not be so straightforward. ★

Finally, we propose an algorithm to translate the ER schema into the corresponding schema category. In other words, the expressive power of our approach is at least comparable to the ER model. In fact, our approach even extends ER as it represents properties of other logical models (i.e., not only relational, but also array, graph, RDF, key/value, document, and column) at the conceptual level (challenge C2, challenge C5).

Extension towards Data Migration

In the course of proposing the data migration algorithms, we decided to extend the internal representation of objects and morphisms of the schema category and added dual (not necessary inverse) morphisms to the so-called *property morphisms*. The internal representation of schema category objects has been extended to $(key, label, superid, ids)$, where elements *key* and *label* have been added. The former is represented as $key \in \mathbb{N}$ and allows unambiguous identification of the object $O \in \mathcal{O}_\mathbf{S}$. The latter allows the object to be assigned a name, which, unlike the previous proposal, now need not be globally unique.

Morphisms are modelled internally as $(signature, dom, cod, min, max)$. Compared to the previous design, *signature*, *dom* and *cod* are added. The *signature* assigns ϵ to identity morphisms, $n \in \mathbb{N}$ to base morphisms, and concatenation of *signatures* of morphisms being composed separated by '.' (dot) to compound

morphisms. The pair *dom*, *cod* represents the domain and codomain of the morphism. The new features allow morphism identification as follows: (1) an identity morphism having $signature := \epsilon$ is determined by the pair dom, cod, since each object can have just one identity morphism, and (2) a non-identity morphism is identifiable only by *signature*, as this value is unique for non-identity morphisms. Note that the $\circ_{\mathbf{S}}$ is also modified to reflect the composition of *signature*, *dom*, and *cod*.

The previous definition of the schema (and instance) category considered dual morphisms only between pairs of objects $O_1, O_2 \in \mathcal{O}_{\mathbf{S}}$ corresponding to classes of real-world objects and their relationships. The new definition of schema category considers dual morphisms even when O_1 or O_2 correspond to a property object. The reason for this change is to allow arbitrary (directed graph) traversal of the schema category, which we exploit during data migration and it is also a preparation for querying based on pattern matching.

In addition, we introduce a concept of *mapping between logical and conceptual layers*, specified for each kind κ by $\mathfrak{M}_\kappa := (\mathcal{D}, name_\kappa, root_\kappa, morph_\kappa, pkey_\kappa, ref_\kappa, P_\kappa)$, where $\mathcal{D}$ denotes a particular database component, $name_\kappa$ represents the name of the kind κ, $root_\kappa$ or $morph_\kappa$ refers to the root property of κ, $pkey_\kappa$ describes the structure of its identifier, ref_κ captures the references from κ to κ' (challenge L4), and P_κ captures the hierarchical structure along with the model-
★ specific constructs of the logical representation of kind κ (challenge L2) (see Subsection 3.3.1 for more details). Hence, the mapping allows us to decompose the schema category into logical units corresponding to each kind κ (challenge L5, challenge L1). Note that by the addition of the mapping of the logical to the conceptual layer, the approach is also ready to extend the support for additional data models (challenge L7).

The mapping, together with the extended schema category, forms the foundation of universal algorithms of data transformation from logical to categorical representation and vice versa. Moreover, these algorithms allow data migration from any input to any output combination of logical models, i.e. the multi-model to multi-model migration. Finally, also a uniform approach to missing data, even though they may be represented differently at the logical level, is supported (challenge L3).

A complete solution is implemented as the open-source academic prototype
★ *MM-cat* [64]. This tool allows a multi-model schema to be represented by a graph freely generating a schema category (see Definition 5). In addition, the tool allows the schema to be automatically extracted from a conceptual model (e.g., ER and UML). Naturally, the tool provides a unified approach to logical models (including their specific properties) by mapping the logical model to a schema category. Finally, *MM-cat* implements transformation algorithms implementing data migration. For proof-of-concept purposes, the prototype supports PostgreSQL (relational and document model, i.e., multi-model representative), MongoDB (document model representative) and Neo4j (graph model representative). The specific properties of the database systems are implemented using
★ so-called wrappers (see Subsection 3.5.1 for more details).

0.4 Schema Inference

An attractive feature of a majority of NoSQL databases is the possibility of storing data in kinds without a previously defined schema. From the user's perspective, this is a simple and flexible way of storing data. However, in various use cases we still require the knowledge of an explicit schema, e.g., in the case of querying, migration, and data evolution. In such cases, it is necessary to infer the non-existing schema secondarily, from the already stored data.

Currently, there are a number of approaches for schema inference over a single data model, mostly for aggregate-oriented database systems based on JSON and XML document models. However, these approaches often suffer from various drawbacks, e.g., the inferred entities contain too many (optional) properties (i.e., the schema is not very clear to users), the inferred schema does not respect the order of the properties (e.g., approaches that infer the schema of JSON documents rarely consider the order of the elements in a JSON array), or they infer integrity constraints in a very limited way or not at all. Moreover, the inferred schemas are often complicated also from the data modelling perspective. For example, the UML [32] does not allow to model an inferred schema if it contains a union type. Therefore, also new schema description formats have emerged, though often non-standardised, tailored to a single particular system, or supported only by a particular schema inference approach (e.g., Baazizi's proprietary language [65]), making it difficult to compare schemas inferred by different approaches or even over different database systems. Hence, we aim to thoroughly analyse the problem of schema inference over multi-model data and verify whether it is possible to extend any of the existing approaches in order to infer a schema for multi-model data, or if a completely new approach is necessary.

In this section, we first describe the state-of-the-art approaches to schema inference over individual data models. We then elaborate on five recent JSON schema inference approaches and compare them statically to answer the question of whether any of the selected approaches can be used to infer a schema of multi-model data. Based on the analysis of the selected schema inference approaches, we discuss a set of open questions and issues in schema inference in the context of multi-model data. Finally, we present our fully multi-model schema inference approach.

0.4.1 State of the art

The research on inferring the implicit schema of data is not new. It includes not only modern single-model NoSQL databases, but also older technologies such as XML and RDF [66]. The principles of the different algorithms are similar, differing only in the features supported by each format, e.g., capturing the order of properties in case of XML. In addition, the approaches are often scalable and support parallel processing of Big Data.

Graph and Linked Data Schema Inference There are several approaches dealing with schema inference in the graph model and Linked Data. Lbath et al. [67] focus on inferring simple and complex types in a property labelled graph, including its hierarchy and cardinality of edges. Galinucci et al. [68] address

schema inference for RDF documents, where they identify aggregated hierarchies and repeating patterns in Linked Open Data. Finally, Bouhamoun et al. [69] tackle the horizontal scaling problem of processing large amounts of RDF data and present a method based on pattern extraction in linked data.

Key/Value Schema Inference The problem of inferring schema in key/value stores and then transforming the data into (flat) relational tables is studied by DiScala and Abadi [70]. In this case, the authors work with systems that store structured JSON documents in place of values, but treat them as black boxes at the database system level.

XML Schema Inference There are several comprehensive papers dealing with approaches to inferring schemas in XML documents [71]. A comparison of existing heuristically based approaches, including open problems, is provided in [72]. These are primarily older approaches, popular before the advent of a more popular format – JSON. A comparison of the grammar inferring approaches can then be found in [73].

Heuristic approaches [74, 75, 76, 77, 78, 79] are based on generalising the schemas of individual XML documents based on a set of predefined heuristic rules. These methods can be further subdivided according to the chosen strategy: (1) The approaches [75, 76, 78] gradually generalise the schema until they reach the desired solution. (2) The approach [77] generates a large number of candidates and selects the most suitable schema. (3) Algorithms called *merging state* [76, 78] are based on searching a heuristically selected subspace of all possible schema generalisations of a given XML document. They represent the schemas as states of a prefix tree automaton and construct sub-optimal solutions by merging its states.

Alternatively, there are methods based on *grammar inference* [80, 81, 82, 83, 84, 85, 86]. These methods consider an XML schema as a grammar and the XML document corresponding to the schema is the word generated by the grammar. Moreover, this problem can be reduced to the extraction of a set of regular expressions, where one regular expression describes one XML element. Moreover, the approaches exploit additional information besides the XML documents, e.g., a predefined maximum number of nodes of the target automaton, since according to Gold's theorem [87], regular languages cannot be identified based on positive examples alone.

The majority of the existing approaches represent the resulting schema using the DTD language. Only the approaches [86, 78, 88] represent the schema using the XML Schema language.

JSON Schema Inference Although the JSON and XML document formats are very similar (i.e., both are semi-structured hierarchical data formats), schema inference approaches for XML documents are often not applicable to large collections of JSON documents [16], not only because of the differences between these formats (e.g., XML is order-preserving and duplicate-allowing while JSON is order-ignorant and duplicate-prohibiting), but also because the existing schema inference approaches for XML documents do not assume large data collections (i.e., Big Data) and their scalability and parallelisability are thus limited.

★

A comparison of the static properties of selected JSON schema inference approaches is addressed in works [89, 90]. At the same time, popular approaches are also described in our work [16], where we additionally investigate the applicability of selected approaches for schema inference over multi-model data. Finally, the comparison of dynamic properties is addressed in our paper [91]. ★ ★

Sevilla Ruiz et al. [92] propose an approach of inferring versioned schemas from document-based NoSQL databases. The foundation of the approach is an abstract model based on the *Model-Driven Engineering* (MDE). This work is followed up by Chillon et al. [93], who address the visualisation of NoSQL database schemas and propose extensions needed to visualise aggregate-oriented data. Most recently, Fernandez et al. extend the abstract model by adding the support for relational and graph data [30].

Klettke et al. present a complex solution for managing NoSQL schemas [94], including an approach for reconstructing schema evolution history in so-called *data lakes* [95]. This research is followed by a tool *jHound* [96] that enables profiling of JSON data, e.g., searching for structural outliers. Finally, the tool *Josch* [97] allows schema extraction from JSON data, schema refactoring, and subsequent validation against the original dataset.

Baazizi et al. [65] propose a horizontally scalable approach for parameterised schema inference from large collections of JSON documents. They also introduce a custom and compact language for describing the resulting JSON schema.

Izquierdo and Cabot [98] proposes an approach to infer schemas for web services based on JSON documents. The authors also provide a web application along with a visualisation tool [99].

An approach of inference of schema over collections of JSON documents is also presented by Frozza et al. [89]. In contrast to previous works, the authors consider inference of data types specified by the BSON standard.[26] Unfortunately, their approach has limited parallelisability.

Last but not least, Wang et al. [100] propose a schema inference method over document repositories based on finding equivalent subtrees (i.e., frequently repeated hierarchical structures).

The JSON Schema language is primarily used to describe the inferred schema of JSON documents. The formal model of this language is discussed by Pezoa et al. [101], while the general description of the JSON type system is discussed by Baazizi et al. [102].

Columnar Schema Inference Frozza et al. have also proposed an approach for schema inference over columnar NoSQL databases [103], specifically supporting the inference of implicit schema from the *HBase*.[27]

Summary There is a number of approaches aimed at inferring a schema over a particular data model. Unfortunately, to the best of our knowledge, there is currently no approach applicable to infer a schema over a combination of data models, i.e., multi-model data. At first sight, it may seem that existing single-model approaches can be also applied to infer a multi-model schema, but as we will show in the next subsection, this idea is not feasible in practice, as the individual

[26] https://bsonspec.org
[27] http://hbase.apache.org

data models often have contradictory features. Moreover, the combination of models itself is a complication, as we must additionally consider references and redundancy across models, not just within a single data model.

0.4.2 Closely Related Single-Model Approaches

The utilisation and extension of verified single-model approaches seems promising for the inference of a multi-model schema. To validate this idea, we chose inference approaches over JSON documents because the JSON format is sufficiently complex to cover a variety of data constructs, at least at first sight. Moreover, these approaches are often horizontally scalable and thus covering also high volumes of data.

In particular, we focus on five selected schema inference approaches, namely:

1. The approach proposed by Sevilla Ruiz et al. [92] working with the concept of distinct versions of entities.

2. The approach of Klettke et al. [94] using a graph structure to represent a schema and able to detect outliers.

3. The approach of Baazizi et al. [65] which introduces a comprehensive and massively parallelisable method for inferring of schemas.

4. Izquierdo and Cabot [98] approach which can infer a schema from multiple document collections.

5. The approach of Frozza et al. [89] which is able to infer schemas including data types as introduced in BSON.

We compare the approaches statically, i.e., we focus mainly on their basic principles and algorithm scalability, input and output parameters, possible support for structural components beyond the JSON format, distinguishing between optional and required properties, support for inference of integrity constraints, and detection of redundancy in the data. We also verify whether the selected approaches are applicable to infer a multi-model schema. Table 0.4 summarises the comparison of key characteristics of the selected approaches. (A comparison with our approach is also included; however, our approach is not discussed in the following paragraphs. It is introduced in Chapter 4.)

★

Inference Process The majority of approaches generate a schema based on all documents in the input collection. An exception is the approach of Izquierdo and Cabot which retrieves documents from web services. Moreover, the approaches of Sevilla Ruiz et al. and Frozza et al. minimise the input document collection into a collection that contains only structurally distinct documents. A common feature of all approaches is the replacement of property values by the names of the supported primitive data types they encounter.

Scalability Design Based on the theoretical design, most approaches are horizontally scalable. Unfortunately, their implementations are usually not parallelised. Sevilla Ruiz et al. use MapReduce in order to select structurally distinct documents from the input collection. Unfortunately, in the worst case, where all documents are structurally distinct, the scalability of this approach is limited since the subsequent processing of the minimal collection of structurally distinct documents (i.e., in this particular worst case, the entire input document collection) is not parallelised. In contrast, the approach of Baazizi et al. is fully horizontally scalable because Apache Spark is used throughout the schema inference process.

Implementation The approaches of Sevilla Ruiz et al. and Izquierdo and Cabot are Java applications running in the platform-independent Eclipse environment. Klettke et al. implement their approach as a Spring Boot application (i.e., as a platform-independent application built in Java) and additionally allow the approach to be deployed as a Docker container. Baazizi et al. implement the approach using Scala as an Apache Spark job. Finally, Frozza et al. approach is implemented as a javascript web application.

Input All approaches support schema inference from a collection of JSON documents. Frozza et al. approach also supports BSON data. In addition, the approaches of Sevilla Ruiz et al. and Klettke et al. allow arbitrary aggregate-oriented data to be converted to JSON data, and thus apply their approach for schema inference over these models as well. However, by converting data from other models to JSON documents, we may lose structural information, e.g., because JSON does not preserve the order of features and does not allow explicit representation of complex data structures such as maps, sets, and tuples. Hence, a schema inferred this way may not be accurate.

Moreover, only the approaches of Sevilla Ruiz et al. and Izquierdo and Cabot allow the inference of a schema from the entire input database, i.e., from a set of collections. The remaining algorithms infer schemas only over individual collections.

Output The approaches represent the inferred schema by means of textual or graphical languages. Klettke et al. and Frozza et al. approaches use the JSON Schema language to describe the schema, although they differ in the details of use. Baazizi et al. represent the schema with their own compact but complex proprietary language. Finally, Sevilla Ruiz et al. and Izquierdo and Cabot represent the inferred schema as a UML class diagram.

Structural Components The selected approaches differ in the extent to which they support the inference of different structural components of JSON documents. Most approaches are only able to infer a basic set of primitive types (i.e., *String*, *Number*, and *Boolean*) and some complex types (i.e., nested objects and arrays). Whereas, of the approaches we observed, only Frozza et al. allows the inference of the extended data types introduced in BSON. Unfortunately, the approaches do not allow distinguishing other complex data structures, namely maps, sets, and tuples, as even the JSON format does not distinguish these structures from nested

structured objects (also representing, e.g., maps) and arrays (also representing, e.g., sets and tuples).

Optional Properties The majority of approaches distinguish between required and optional properties, differing only in the way they are detected. The approaches of Klettke et al. and Frozza et al. compute differences in the occurrence of individual properties compared to the occurrence of parent properties. If the parent property occurs more frequently, the child is marked as optional. Sevilla Ruiz et al. are able to detect optional properties using set operations over entity versions (intersection of properties of two or more entity versions returns required properties, while union of entity versions minus their intersection returns a list of optional properties). Baazizi et al. detect optional properties during the merging of two document schemas. Finally, Izquierdo and Cabot is the only approach unable to distinguish between optional and required properties. In this case, we consider all properties as required.

Union Type The majority of the compared approaches work with the concept of union type. The approaches of Klettke et al. and Frozza et al. use the JSON schema keyword *oneOf*, while Baazizi et al. represent a union type as a concatenation of types with the "+" character. Only the approaches of Sevilla Ruiz et al. and Izquierdo and Cabot do not consider the concept of union types. The former consider versions of entities where union types cannot naturally occur. As for the latter, the type of a property is expressed only by the most general of the identified types, e.g. as *String*.

Order Preserving All approaches treat JSON documents as a set of unordered properties and therefore do not consider order detection. Unfortunately, the order is not even considered in the case of array elements, where the order matters.

Integrity Constraints Integrity constraints are detected only to a limited extent or not at all. Identifiers (whether simple or complex) are not detected by any of the approaches. References are only partially detected in Sevilla Ruiz et al. approach, based on a naming convention. The inference of other (complex) integrity constraints is not considered at all, e.g., ranges of values of individual properties or mutual dependency between values of different properties.

Data Redundancy Although redundancy in data is a common feature of NoSQL document stores, most of these approaches do not detect this feature. Only the approach of Izquierdo and Cabot allows merging the schema of two collections (i.e., considering them as redundant) if they have identically named properties, but does not verify this fact at the data level.

Additional Features To conclude the comparison, let us also focus on a couple of specific features supported by just some of the covered approaches. In particular, the approach by Sevilla Ruiz et al. allows for the visualisation of the inferred schema [93], including the visualisation of entities and relationships using UML. The approach by Klettke et al. allows us to analyse documents using a

proprietary profiling tool *jHound* [96]. The approach also detects errors and outliers, i.e., additional or missing properties. To continue, the approach by Baazizi et al. is suitable for inferring schemas over large JSON document collections due to the massive parallelisation applied. Fusion properties, i.e., associativity and commutativity, enable the evolution of an already inferred schema, making this approach suitable for collections of documents that are rapidly evolving. The approach by Izquierdo and Cabot is designed to generate schemas for sets of web services producing JSON documents with similar features. With a global schema, the user then gets an idea of which specific services to call to get the requested information. A part of this approach is also a schema visualisation tool [99], which produces a schema compliant to JSON Schema. Finally, the approach by Frozza et al. is designed to work over the *MongoDB* database and supports the extended BSON format.

As illustrated in Table 0.4, the approaches do not consider natural features of multi-model data, e.g., variety of data types, complex structures, ordering of properties and theirs duplicities, identifiers, inter- and intra- model references and redundancy, and complex integrity constraints in general. Thus, despite the complexity of JSON model, the strategy of converting multi-model data into JSON data and inferring its schema is not sufficient as we lose critical schema information.

0.4.3 Open Questions and Challenges in Schema Inference

We believe that in order to be able to infer schema even for multi-model data, we need to address the limitations of existing approaches and extend them appropriately. Therefore, we provide the following list of **challenges I1 − I10** in the area of schema inference for multi-model data.

- **I1:** *Integrity constraints.* Generally, the schema not only describes the structure of the data, but it may include a list of integrity constraints, e.g., identifiers, references, ranges of property values, and rules describing complex dependencies between properties. At first sight, this challenge goes beyond the bounds of some data models and the languages describing their schema. For instance, JSON and JSON Schema do not allow modelling complex integrity constraints. However, this does not mean that we do not have implicit integrity constraints in JSON data. In the case of multi-model data, we could use, e.g., OCL [48] to describe integrity constraints if the underlying data model does not explicitly support them.

- **I2:** *Values of properties.* Most of the observed approaches infer the schema only from the structure of the data, i.e., the names of the features and mutual hierarchy. The values are only converted to their data types and further processing of the values themselves is omitted. We believe that, e.g., a statistical analysis of property values can refine the inferred schema, e.g., allowing the inference of identifiers, references, and some other integrity constraints.

- **I3:** *Multiple data models.* A multi-model schema is a union of schemas from data represented by various logical models. Underlying data models may

Table 0.4: Comparison of the selected schema inference approaches

	Sevilla Ruiz et al. [92]	**Klettke et al.** [94]	**Baazizi et al.** [65]	**Izquierdo and Cabot** [98]	**Frozza et al.** [89]	**MM-infer** [4]
Inference Process	MapReduce + MDE	Fold into graph	Reduction in Apache Spark	MDE	Aggregation + fold into graph	Aggregation in Apache Spark
Scalable design	Yes	Yes	Yes	Yes	No	Yes
Scalable implementation	Yes	No	Yes	No	No	Yes
Implementation	Eclipse bundle	Spring Boot application	Apache Spark application in Scala	Eclipse bundle	Node.js web application	Apache Spark application in Java
Input format	Aggregate-oriented NoSQL data	JSON	JSON	JSON web service responses	Extended JSON	Multi-model data
Input type	Multiple kinds	Single kind	Single kind	Multiple kinds	Single kind	Multiple databases
Output format	NoSQL Schema model	JSON Schema	Custom textual type language	Ecore model	JSON Schema	ER, UML, JSON Schema, XML Schema, Categorical schema
Schema root	Entity	Record	Record	Entity	Record	Property
Extended JSON	No	No	No	No	Yes	Yes
Tuple	No	No	No	No	No	Yes
Set	No	No	No	No	No	Yes
Map	No	No	No	No	No	Yes
Optional	Yes	Yes	Yes	No	Yes	Yes
Union type	No	Yes	Yes	No	Yes	Yes
Order preserving	No	No	No	No	No	Yes
Identifiers	No	No	No	No	No	Yes
References	Partial	No	No	No	No	Yes
Complex IC	No	No	No	No	No	Partial
Data redundancy	No	No	No	No	Partial	Yes

also arbitrarily overlap (intersect), i.e., some data are partially or completely redundant. Moreover, we combine models that often have contradictory features and are based on different principles, e.g., aggregate-ignorant/oriented, order-ignorant/preserving, or allow different types of identifiers and references. Finally, we have to consider not only the features of the particular models, but other features resulting from their combination, e.g., cross-model references and cross-model data redundancy. All this brings a new dimension of complexity to the problem.

I4: *Unification and abstraction of complex data types.* At first sight, popular data models seem to approach common data structures (i.e., tuple, list, set, and map) in a uniform way. A closer look shows that this is not the case. For example, the column model explicitly distinguishes between all these structures, whereas the JSON document model explicitly considers only a list and a nested object. The tuple and set are implicitly represented by a list and the map as a nested object naturally containing mainly optional properties. In order to be able to infer a multi-model schema, we need not only the unification of the corresponding constructs across data models, but also their appropriate and not too general abstraction.

I5: *Scalability.* The proposed approach should be able to handle large volumes of multi-model data, but the inability to scale horizontally is a limitation of many observed approaches. Moreover, the approaches work with a large logical unit of data at a time. In particular, in one step the algorithms merge two records. If the data contains a large number of optional properties, the merged record is always more complex and an increasingly complex schema is continuously propagated to the next stages. Conversely, if we are working with a suitably small logical unit, e.g., merging schemas at the level of individual properties, then the large number of optional properties in the data has no effect on the structure of the merged single-property schema. Hence, algorithms working with a smaller logical unit of data can be significantly more efficient and scalable.

I6: *Ordering of properties.* The order of properties is a natural feature of several popular data models. For example, we can consider the order of elements in an XML document or the order of elements within a JSON array. Since multi-model data combines the features of each data model, the schema inference algorithm should be able to infer the order of features and elements.

I7: *Data redundancy*, i.e., the ability to represent the same data by various logical models, is another feature typical for multi-model data. A potential redundancy inference could improve query performance, e.g., to enable alternative query evaluation strategies.

From a more general point of view, the problem of inference of a multi-model scheme also involves the following open questions:

I8: *Fetching data.* Currently, schema inference approaches are tightly bound to a particular database system that implements a particular variant of the data model. As a result, the approaches may not be directly applicable

for schema inference in another system that supports the same data model. In addition, existing approaches often access data in a specific way. An optimal algorithm should be independent of the chosen database system and should also allow for different ways of retrieving data.

I9: *Universal processing.* The foundation of an ideal algorithm should be system-independent, e.g., exploiting unification of data model constructs. However, this universal algorithm can be based on so-called *wrappers* that implement the features of individual database systems and convert the input of the algorithm, e.g., structural information of data, into a unified form.

I10: *Schema representation.* Various graphical or textual languages are used to represent the (inferred) schema – for example ER [31], UML [32], DTD [71], XML Schema [104, 105], JSON Schema [106] and others. However, these formats are insufficient for representing the schema of multi-model data because they do not reflect all of its structural features.

0.4.4 Contribution: Framework MM-infer

So far, we have discussed the related work, the applicability of verified schema inference approaches to multi-model data, and the resulting open questions. We now turn to a commentary on our approach and explain the connection to the rest of the project.

At first sight, the JSON format appeared to be comprehensive enough to cover the structural features of multi-model data. At the same time, schema inference approaches over the collections of JSON documents scale very well and are capable of handling Big Data, and therefore seem like good candidates for extensions towards multi-model schema inference. However, during our research of existing ★ approaches [16], we found out that a selected single-model approach will not only need to be extended but also combined with features of other approaches.

In the thesis of Ivan Veinhardt Latták [107] (supervised by Pavel Koupil), we tried to extend a chosen approach to support multi-model data. In particular, the work resulted in the design of an algorithm based on the approaches of Baazizi et al. [65] and Sevilla Ruiz et al. [92]. However, not even this algorithm is sufficient for schema inference for multi-model data. Hence, based on the identification of the drawbacks of the proposed approach, an extended list of requirements that an ★ optimal schema inference algorithm should satisfy was created [91]. In addition, the paper also experimentally compares the dynamic features of the approaches from Table 0.4.

The core drawback of the proposed algorithms is the lack of coverage of all structural features of different popular data models. Therefore, we performed a thorough analysis of selected popular data models and introduced a unification of semantically similar features (challenge I4). We found out that the basis of the unified model is the name/value pair. We refer to this pair as a property. Based on the type of the value part, we divide properties into *simple* and *complex*. A simple property has a scalar value, while the value of a complex property can be a (homogeneous or heterogeneous) array, a set, or a map. Note that we only need to consider these three examples if we consider tuples as a special case of arrays

and nested structures as a special case of maps (see Chapter 4). The unification of ★ the constructs allows us to better grasp the general features of multi-model data, and we are able to introduce universal data structures to describe the schema and data, as well as a general horizontally scalable algorithm that considers the varied features of data models from the very beginning and infer the true multi-model schema (challenge I3).

The foundation of our approach consists of two data structures – the *Record Schema Description* (RSD) and the *Property Domain Footprint* (PDF) (see Subsection 4.4.5). ★ The RSD describes the structure of a single property (or a record as a special case of a root property) including the name of the property, the frequency of occurrence, the relationship to other properties within the same record (i.e., the parent/child hierarchy), the order of the children properties (challenge I6), and other features. The second data structure, PDF, allows us to describe all values of a given property in a compact way, including, e.g., uniqueness, multiplicity, occurrence of a particular value etc. This allows us to efficiently compare the active domains of two properties (challenge I2).

Altogether, we proposed three universal algorithms for inferring a structure or integrity constraints in multi-model data (challenge I9):

- The *Record-Based Algorithm* (RBA) infers a schema by gradually merging RSDs that describe the schema of a record.

- The *Property-Based Algorithm* (PBA) was designed to test the hypothesis that processing smaller logical units of data allows for a more scalable approach. We validated our hypothesis using experiments (see Section 4.6) ★ and we confirmed that PBA is much more suitable for dealing with larger volumes of highly structured data (challenge I5), while RBA is more suitable for small volumes of flat data (i.e., aggregate-ignorant systems).

- The *Candidate Miner* extracts candidates for basic integrity constraints, e.g., identifiers, references, ranges of values (challenge I1), and in addition allows us to detect redundancy in the data (challenge I7). Note that we only retrieve candidates, because we describe the active domain using, among other things, Bloom filters, which with certain probability return a false positive result [108]. The user can then validate the selected candidates or the candidates are verified by an algorithm.

All listed algorithms are implemented using the Apache Spark framework and are therefore horizontally scalable.

Finally, the algorithms were verified using the prototype implementation *MM-infer* [109],[28] which currently supports schema inference for data stored in *PostgreSQL* (a representative of a multi-model schema-mixed database), *MongoDB* (a representative of a document schema-free database), and *Neo4j* (a representative of a graph schema-free database). Access to each database and retrieval of the data from which the schema is subsequently inferred is done via system-specific wrappers (challenge I8). And the tool also allows the selection of the format for representing the resulting inferred schema (challenge I10).

[28] https://www.ksi.mff.cuni.cz/~koupil/mm-infer/index.html

0.5 Evolution Management

Despite the correctness of the database schema design, sooner or later user requirements may change. A process that reflects the change of user requirements into the schema, data and related queries, integrity constraints, and data storage strategy while maintaining the integrity of the overall system is referred to as *evolution management*. Currently, this represents one of the most complex challenges [110].

Evolution management involves three main tasks: (1) managing structural changes of data, either explicitly applying *schema modification operations* (SMOs) or implicitly by reverse engineering, (2) propagating structural changes to data, i.e., data migration strategies, and (3) propagating such changes to queries. Moreover, there are, e.g., benchmarks that evaluate the effectiveness of data migration [111] and querying operations [112, 113].

Currently, there is a number of evolution management approaches for relational DBMS [20, 114, 115] or NoSQL systems [110, 116, 117, 118, 119] and the first approaches proposed for multi-model DBMS [120, 121] are emerging. However, the existing solutions have various limitations, targeting only a small fraction of data models [122], and partially or completely ignoring integrity constraints and features arising from the combination of multiple data models [117].

In this section, we elaborate on five selected promising approaches to evolution management. Based on the analysis of the selected approaches, we discuss a set of open questions and challenges in evolution management of multi-model data. Finally, we present our evolution management approach.

0.5.1 Closely Related Approaches

We first review two approaches [19, 20] that build an evolution management approach on an abstract data model. Next, we discuss approaches applicable to NoSQL database systems [110, 120, 116] that, in addition to schema changes and their propagation to the data, also consider other aspects of evolution management. Finally, we mutually compare the selected approaches.

Evolution Management in CGODD

The idea of utilising category theory in evolution management is not new. The beginnings can be found, e.g., in the academic approach [19] (see Subsection 0.3.2). Let us recall that (1) the approach is based on the notion of a *typegraph*, representing a schema and corresponding to an object T of a functor category $\mathbf{G}$ (see Lemma 1), (2) the data is represented by a graph corresponding to an object G of a functor category $\mathbf{G}$, and (3) a data instance is a morphism $Inst : G \to T$ associating data with the schema.

In this approach, schema changes are implemented using a basic set of (SMOs) based on graph pattern matching as follows:

- The *single addition* operation, denoted `ADD_S`, retrieves a single part of the data G corresponding to the pattern P_T and extends it according to the pattern Q_T. Categorically speaking, this corresponds to a pushout (see Definition 17) from $Q_T \xleftarrow{f} P_T \xrightarrow{m} G$, where $m : P_T \to G$ is a graph pattern

matching and morphism $f : P_T \to Q_T$ represents the intended modification. Note that the pattern Q_T may describe a more complex extension, not just an addition of a single property.

- The *full addition* operation, denoted `ADD_F`, finds all occurrences of the pattern P_T in the data G and extends them according to the pattern Q_T. Categorically speaking, this is the construction of a limit [19, 123].
- The *single deletion* operation, denoted `DEL_S`, finds a single part of the data G corresponding to the pattern Q_T and preserves only the subpart corresponding to the pattern P_T.
- The *full deletion* operation, denoted `DEL_F`, finds all occurrences of the pattern Q_T in the data G, and preserves only the data corresponding to the pattern P_T. Categorically speaking, this is again a construction of a limit.

Example 0.14. Figure 0.12 illustrates an example of an extension of a particular address in data G by the vertex named "Czechia", i.e., a single addition operation. The intended modification of the data is represented by the morphism $f : P_T \to Q_T$, i.e., we join edgewise the vertex named "Czechia" to the pattern P_T (indicated in green). Graph pattern matching is represented by the morphism $m : P_T \to G$ (indicated in blue). Note that in this particular case there are two morphisms $m, m' : P_T \to G$, each mapping the graph pattern P_T to a different part of the data G (note that only the morphism m is illustrated). Finally, the data extension is categorically implemented as a pushout from $Q_T \xleftarrow{f} P_T \xrightarrow{m} G$ (indicated in red), i.e., we extend the data from G corresponding to the pattern P_T by Q_T, thus obtaining the modified data G'. □

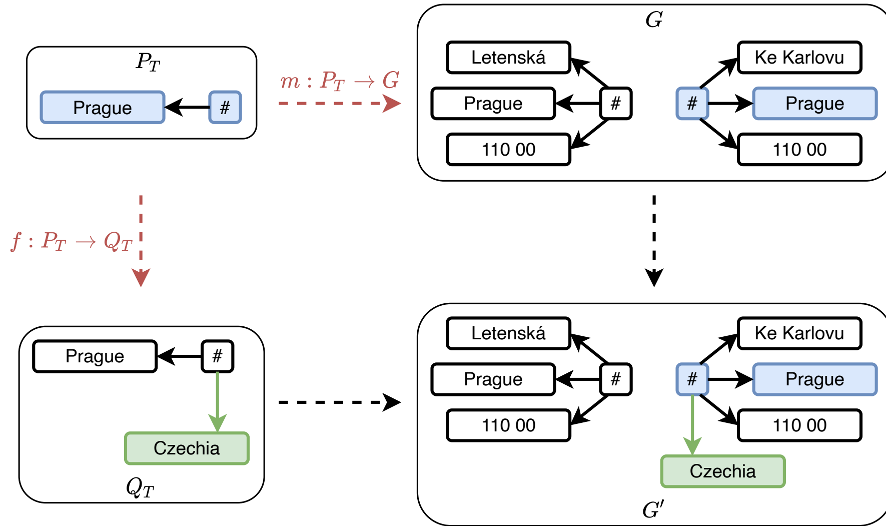


Figure 0.12: An example of a single addition (i.e., a pushout)

The schema changes are immediately propagated to the data (i.e., a strategy commonly known as an *eager data migration*). Note that the SMOs can be combined and concatenated, resulting in complex operations expressing, e.g., union and intersection. Finally, the concatenation of operations can also represent projection, join, and difference, hence the same idea is also applicable to querying based on graph pattern matching.

Evolution Management of Relational Data (Spivak et al.)

Yet another academic approach [20], this time relying on the logical model introduced in Subsection 0.3.2, allows relational data migration according to schema changes. Moreover, the approach enables a change of data representation at the logical layer.

Let us recall that the schema is represented as a free category $\mathbf{C}$ (see Definition 5) and the corresponding data instance is given by the set-valued functor $Inst_{\mathbf{C}} : \mathbf{C} \to \mathbf{Set}$. In addition, there is a category of all instances corresponding to the schema $\mathbf{C}$, i.e., the functor category $\mathbf{Set}^{\mathbf{C}}$ (see Lemma 1 and Example A.8).

In this approach, an SMO translating the schema $\mathbf{C}$ into $\mathbf{D}$ categorically corresponds to the functor $F : \mathbf{C} \to \mathbf{D}$. These functors can be used to represent operations, e.g., rename, delete, copy, intersection, and union for both kinds and properties. Note that the approach does not implement the operation addition, but it only introduces SMOs over existing data.

The schema changes, represented by the functor $F : \mathbf{C} \to \mathbf{D}$, are then propagated to the data utilising the so-called *data migration functors*:

- The functor $\Delta_F : \mathbf{Set}^{\mathbf{D}} \to \mathbf{Set}^{\mathbf{C}}$ propagates operations rename, delete, and copy to the data at the level of a kind or a property. Note that although the functor $F : \mathbf{C} \to \mathbf{D}$ transforms $\mathbf{C}$ into $\mathbf{D}$, the functor Δ_F propagates changes in the opposite direction (i.e., pointing "backwards"). Categorically, this is the principle described as "pulling back along functor F" and functor Δ_F is referred to as a *pullback* [20].[29]

- The functor $\Pi_F : \mathbf{Set}^{\mathbf{C}} \to \mathbf{Set}^{\mathbf{D}}$ propagates operations of intersection and (again) renaming of both a kind or a property. Categorically speaking, the functor Π_F corresponds to the right adjunct [20, 123] of the pullback functor Δ_F and is referred to as the *right pushforward*.

- The functor $\Sigma_F : \mathbf{Set}^{\mathbf{C}} \to \mathbf{Set}^{\mathbf{D}}$ propagates operations of union and (once again) renaming of both a kind or a property. Categorically, the functor Σ_F corresponds to the left adjunct [123] of the pullback functor Δ_F and is referred to as the *left pushforward*.

Example 0.15. Figure 0.13 illustrates an example of a schema change represented by the functor $F : \mathbf{C} \to \mathbf{D}$ (see Figure 0.13 (a)), i.e. merging kinds Audiobook and Book into a single kind Product.

Pullback functor Δ_F (see Figure 0.13 (b)) performs copy, delete and rename operations, i.e., the data from kind Product is first copied into two new kinds, the kinds are renamed to Audiobook and Book, and finally a property Pages is removed from Audiobook and a property Length is removed from Book.

Right pushforward Π_F (see Figure 0.13 (c)) implements the intersection and rename operations. First, a new kind is created by the intersection of Audiobook and Book,[30] and subsequently the newly created kind is renamed to Product.

The left pushforward Σ_F (see Figure 0.13 (d)) implements the union and rename operations. First, a new kind is created by merging Audiobook and

[29] Note that the pullback is an overloaded notion in category theory. It denotes not only the pullback introduced in the Definition 16, but also other constructs [123].

[30] Note that the identifiers of the corresponding records are merged.

Book, and then the new kind is renamed to Product. Note that left pushforward performs a so-called *skolemization* [124], which can be roughly understood as utilisation of null meta-values. □

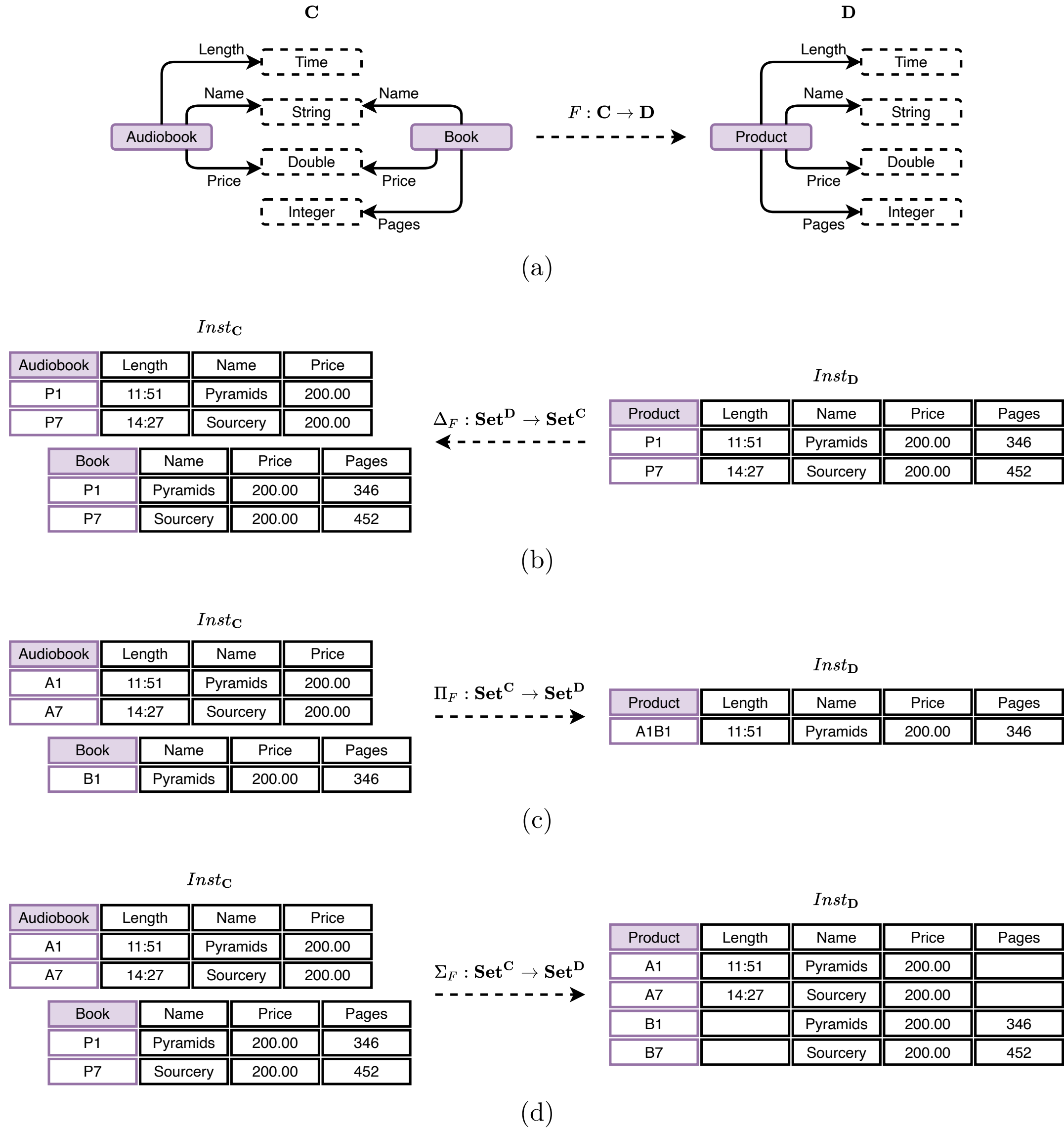


Figure 0.13: An example of schema mapping functor (a) inducing pullback (b), right pushforward (c) and left pushforward (d), i.e., data migration functors

The approach also allows to change the logical representation of data, in particular from relational model to RDF data and vice versa. Categorically, the data transformation is based on the application of the category of elements (also known as Grothendieck construction) [20, 123].

Recently, the authors of the paper [57] proposed an extension of the approach [20] to support multi-model data. In particular, the extension allows the representation and migration of data logically represented as JSON documents or graphs. However, as far as we know, this is only a theoretical extension and it is not implemented in CQL.

Darwin

Among the many academic prototypes, Darwin [110, 125] represents a family of approaches targeting schema management and data migrations. Basic features include semi-automatic declaration of schema changes [126], propagation of changes to data [127], inference of versioned schema including the historical sequence of versions [95], and various optimisation proposals [122, 128, 129, 111].

The authors propose a general programming language, the *Schema Evolution Language* (SEL), which allows declaring the following platform-independent SMOs:

- The *add* operation appends a property to every (selected) record of the particular kind or adds a new kind.
- The *delete* operation removes the property from each (selected) record of the particular kind or performs the removal of the kind.
- The *rename* operation changes the name of the property in all (selected) records of the particular kind or renames the kind.
- The *move* operation moves the property from each (selected) record of the input kind to the particular records of the destination kind.
- Similarly, the *copy* operation makes a copy of each (selected) record in a particular kind and inserts the copy into the particular record of the destination kind.

In addition, the operations can be assigned specific values (e.g., an added value in the case of the operation add) or filtering conditions specifying the selection of a subset of records of a particular kind. Also note that SEL expressions are subsequently translated into a domain specific language.

As for the propagation of SMOs to the data, the approach allows for multiple data migration strategies:

- *Eager strategy*: The changes are propagated immediately after the SMO is executed.
- *Lazy strategy*: The migration to a new version is delayed until a new version is required to perform a data management task.
- *Incremental strategy*: A hybrid strategy that completely migrates data, e.g., after a predetermined number of SMOs have been performed. Otherwise, the data is migrated lazily.
- *Predictive strategy*: A hybrid strategy that tracks the number of accesses to data. Frequently accessed data is migrated immediately after SMO execution, while the remaining data is migrated lazily.

The complete solution also includes a set of so-called *composition rules* [130], which allow to compose SMOs. Since typically composed SMOs are executed repeatedly, caching is utilised, which significantly improves the performance of data migration of different versions [131]. In addition, the authors propose a utilisation of composition rules for query rewriting [132].

Finally, the family of approaches includes MigCast [122] and EvoBench [111]. The former tool allows comparing different data migration strategies, e.g. in terms of operational costs, and the latter is a benchmark for schema evolution. The complete set of tools is then available as a docker container,[31] currently supporting a number of NoSQL database systems such as, e.g., MongoDB, Couchbase, Cassandra and ArangoDB.

MM-evolver

The first approach targeting evolution management of multi-model data is the academic prototype MM-evolver [120]. This tool allows propagating of SMOs to data represented by different and interconnected models and additionally considers (in a limited way) cross-model references and redundancy.

The approach implements the set of platform-independent SMOs supported by Darwin, and in addition introduces a special operation to add/delete a reference. Note that internally, the operation differs from the Darwin system in propagation to multiple data models as well as propagation to references. For example, if a property is removed, all references (including intra- and inter-model) are removed as well.

Similar to the Darwin approach, the authors propose a general programming language for declaring platform-independent SMOs, the *Multi-Model Schema Evolution Language* (MMSEL). As in the previous case, MMSEL expressions are subsequently translated into a domain specific languages.

Finally, schema changes are immediately propagated to the data, i.e., only the eager strategy is applied.

Orion

Recently, the family of academic approaches for representation [30] and management [92, 93] of data in NoSQL systems has been extended with the language Orion [116], which allows declaring SMOs over the U-Schema logical model (see also Subsection 0.3.2).

From a logical unit perspective, schema modification operations (SMOs) declared by Orion can be classified as kind-level, version-level, and property-level. Kind-level operations include common operations such as *add*, *delete*, *rename*, *extract* (i.e., copy), and in addition:

- The *split* operation splits the set of properties of one kind to create two new kinds, the original kind being removed.

- The *merge* operation merges the properties of two kinds to create a new kind, replacing the original kinds.

Version-level operations, performed at the level of record version, include:

- The *delvar* operation removes a specific record version from the schema, and the change is propagated to the data by deleting all records corresponding to the deleted version.

[31] https://sites.google.com/view/evolving-nosql/tools/darwin

- The *adapt* operation also removes the record version from the schema, but the corresponding records are converted (adapted) to another, selected record version.
- The *union* operation merges two versions of the records into one of them.

Finally, property-level operations are implemented over simple and complex properties (including references as a special kind of a property), and allow for the common operations such as *add*, *delete*, *rename*, *extract*, and *move*. They also include:

- The *nest* operation allows nesting properties, i.e., moving the selected property to the nested complex property.
- The *unnest* operation, which is inverse to *nest*.
- The *cast* operation allowing to change the data type of a simple property or a reference.
- The *promote* operation implementing the addition of an attribute to the key.
- The *demote* operation, on the other hand, allows the attribute to be removed from the key.
- The *mult* operation implementing a cardinality change on a reference or a complex property (e.g., an array).
- The *morph* operation allows to replace a reference with a nested complex property (aggregate) and vice versa.

As in the previous cases, the Orion language allows the usage of filter conditions to specify the subset of records over which the SMO is performed. Once again, SMOs are platform-independent and translated into domain-specific languages, and the schema changes are propagated eagerly.

Currently, the supported systems include MongoDB, Cassandra, and Neo4j, i.e., representatives of document, column, and graph databases. However, the the U-Schema model allows the extension towards the support of key/value and (a partial support of) relational DBMSs.

Although the authors attempt to create a database-independent language for describing schema changes, the language is burdened by the integration, (but) not the unification of data models in U-Schema. As a consequence, the language is complex as it has to take into account all model-specific constructs and properties of underlying data models. Hence, the extensibility of the approach is limited.

Comparative Summary

Table 0.5 illustrates a comparison of the described evolution management approaches. It also includes a comparison with our proposed approach (MM-★ evocat), but it is addressed in a separate chapter (see Subsection 0.5.3 and Chapter 5). All the compared approaches implement a set of SMOs, yet they differ in the choice of the particular supported operations (see Table 0.6).

Table 0.5: A comparison of features in the selected evolution management approaches

	CGOOD [19]	**Spivak et al.** [20]	**Darwin** [110]	**MM-evolver** [120]	**Orion** [116]	**MM-evocat** [5]
Schema modification	Yes	Yes	Yes	Yes	Yes	Yes
Data migration	Eager	Eager	Eager, lazy, hybrid	Eager	Eager	Eager
Version jumps	No	No	Yes	No	No	No
Propagation to IC	No	No	No	Partial	Partial	Yes
Schema inference	No	No	Tracked, untracked	No	Untracked	No
Query rewriting	No	No	Yes	No	No	No
Benchmarking	No	No	Yes	No	No	No
Self-adaptation	No	No	No	No	No	No
Data models	Relational, object, object-relational	Relational, RDF	Graph, key/value, document, columnar	Relational, graph, key/value, document, columnar	Relational, graph, key/value, document, columnar	Relational, array, graph, RDF, key/value, document, columnar
Multi-model	No	No	No	Yes	No	Yes

Table 0.6: A comparison and classification of supported SMOs in the selected evolution management approaches

	CGOOD [19]	**Spivak et al.** [20]	**Darwin** [110]	**MM-evolver** [120]	**Orion** [116]	**MM-evocat** [5]
Model-level	-	-	-	-	-	add, delete, move, copy
Kind-level	add, delete	delete, rename, copy, intersection, union	add, delete, rename	add, delete, rename	add, delete, extract (copy), rename, split, merge	add, delete, rename, copy, move, group, ungroup
Property-level	add, delete	delete, rename, copy, intersection, union	add, delete, move, copy, rename	add, delete, move, copy, rename	add, delete, move, extract, rename, cast, nest, unnest	add, delete, rename, copy, move, group, ungroup, union, split
Identifier-level	-	-	-	-	promote, demote	addId, dropId
Reference-level	-	-	-	reference	morph	addRef, dropRef
Cardinality-level	-	-	-	-	mult	changeCardinality
Version-level	-	-	-	-	delvar, adapt, union	-

As we can see, the vast majority of approaches only allow for propagating of changes to the data immediately. The exception is Darwin, which allows for other data migration strategies, namely lazy and hybrid. The support for different strategies is reflected in the optimisation capabilities, where once again only Darwin considers optimising data migration by composing SMOs, caching the composed SMOs, and performing lazy data migration efficiently.

The propagation of SMOs to integrity constraints is addressed marginally, with only MM-evolver and Orion addressing references or identifiers in a limited way. Similarly, propagation of SMOs to queries and benchmarking of SMOs are less common features.

Next, only Darwin and Orion allow for inference of versioned schema as an alternative to otherwise explicit schema evolution specified using SMOs. Moreover, in the case of Darwin, the historical sequence of schema versions can be inferred.

Finally, the selected approaches also differ in the extent and support of data models. While the categorical approaches are mainly bounded with aggregate-ignorant models (especially the relational model), aggregate-oriented models are commonly supported in the remaining approaches. However, only the MM-evolver approach considers a set of linked or overlapping data models, i.e., multi-model data. In the other cases, a disjunctive set of models is considered.

0.5.2 Open Questions and Challenges in Evolution Management

We believe that in order to be able to process evolution management for multi-model data, we need to address the limitations of existing approaches and extend them appropriately towards the full support of multi-model data. Therefore, we provide the following list of **challenges E1 – E11** in the area of evolution management of multi-model data.

E1: *Multi-model data.* The aspect of multi-model data brings new dimension of complexity to evolution management approaches. In addition to the model-specific features of (otherwise disjunctive) logical models, we must consider features arising from the combination of the models, such as cross-model references, cross-model embedding, cross-model integrity constraints, redundancy, and inconsistency in data. Moreover, we have to deal with often contradictory features of the models, e.g., aggregate-oriented/-ignorant, schema-full/-less/-mixed, and order-preserving/-ignorant.

E2: *Unification (abstraction) of data models.* Currently, there is a number of approaches that struggle with insufficient unification of underlying logical models. Hence the proposed schema modification language is complicated. It is important to unify, for example, the different approaches to the otherwise corresponding data structures (e.g., JSON object and a map), and to work uniformly with different forms of identifiers (i.e., simple, complex, multiple, and overlapping) and links between objects (e.g., embedding or references).

E3: *Propagation of SMOs to queries.* The propagation of schema changes to the data is only one of many efforts to adapt to changing user requirements.

Naturally, as the schema (i.e., a data structure) changes, the queries must be updated to remain syntactically and semantically correct [132].

E4: *Integrity constraints evolution management.* Currently, there are several approaches to modify the data structure in particular. A schema, however, also includes integrity constraints describing identifiers, references, and, e.g., complex business rules. As far as we know, only several approaches targeting relational DBMSs introduce so-called *integrity constraints modification operations* (ICMO) [114], which allow modification of complex business rules. For non-relational DBMSs, only changes to identifiers or references are supported to a limited extent [120, 116]. In addition, a new dimension of difficulty not observed in relational systems is the implicit management of integrity constraint changes in schema-free data, where integrity constraints must first be inferred and the historical sequence of changes in these constraints must be tracked.

E5: *Propagation of changes to the storage strategy.* Not only a new data structure, but also new queries may reflect the change in user requirements. In multi-model systems, we can respond to this change by adapting the logical representation of the data to maintain/improve the efficiency of query evaluation. However, a minimal number of approaches currently exist that deal with the change in logical data representation [130, 120].

E6: *Extraction of changes.* There is a number of possible methods for retrieving information about changes in user requirements. Most often, these are user-specified changes in the form of SMOs. This approach is particularly useful when dealing with schema-full data. In NoSQL systems, approaches that infer implicit versioned and chronologically ordered schema of data exists. Finally, changes in user requirements and data can be inferred by observing changes in queries. However, none of the solutions is trivial, especially in the case of multi-model data.

E7: *Integration of new data models and data formats.* A significant part of existing evolution management approaches assumes that the logical representation of data is time-invariant. Hence, in practice, we encounter approaches that support only a single data model or a limited set of disjunctive models. If additional data formats are needed, then (1) we need to represent this format by means of supported data formats and models, even if at the cost of relaxing the requirements for efficient data processing, or (2) allow the integration of new data models and formats.

E8: *Benchmarking.* In order to evaluate the effectiveness of the data migration process and to query the successfully migrated data, we need to be able to put a price on each operation and compare them with respect to each other. Currently, there are first drafts of a benchmark for multi-model querying [112, 113] and tools to determine the cost of data migration in polystores and multi-model DBMSs [111].

E9: *Intuitive naming of SMOs.* The contradictory features of underlying logical models in multi-model DBMSs can result in seemingly unexpected propagation patterns into (redundant) data. For example, the ungroup operation

changes the logical structure of data represented by aggregate-oriented approaches compared to the representation of aggregate-ignorant approaches in exactly the opposite way, i.e., in the former case there may be an inlining into the parent property within one kind and in the latter case there will be ★ merging of two kinds (see Example 5.5). Although the naming of the ungroup operation accurately reflects what happens in the aggregate-oriented approach, in the case of the aggregate-ignorant approach the naming is completely non-intuitive.

E10: *Data migration overhead.* Currently, there are several strategies to propagate schema changes to the data. Eager migration propagates schema changes to data immediately, regardless of the current DBMS workload. The lazy strategy propagates changes only when the current schema version is needed, introducing an overhead during (critical) data management tasks, similarly to the hybrid strategies, which combine features of eager and lazy strategies. However, the burden of data migration is still on the datastore regardless of the chosen strategy. The question is whether we must necessarily propagate all changes to the data at the physical level, i.e., stress the database system, or whether changes can only be propagated "virtually".

E11: *Involvement of artificial intelligence.* When a change in user requirements occurs, the user initiates the change in the data structure, e.g., by executing an SMO or by inferring the schema from the new data. However, the choice of the SMO may not always be optimal and reflect all user requirements. We believe that by engaging artificial intelligence to be trained based on user input (e.g., data writes, queries, typical system usage time etc.), we can (1) optimise the choice of the logical representation and the data schema with respect to the data management tasks to be performed, and (2) appropriately plan and select a data migration strategy, e.g., based on the DBMS workload at a specific time.

0.5.3 Contribution: Framework MM-evocat

So far, we have discussed the related work, the applicability of established approaches to evolution management towards multi-model data, and the resulting open questions. We now move on to comment on our approach and explain the connections with the rest of the thesis.

First, we analysed selected existing evolution management solutions introducing a platform-independent layer for dealing with multiple data models [19, 20, 110, 120, 116] and verified their applicability to multi-model data, identifying drawbacks of the selected solutions and outlining open questions.

Being inspired by existing solutions, our schema evolution approach is based on an appropriate abstraction of data models[32] (challenge E1) allowing extension towards the support of additional models (challenge E7), and we define a basic set of SMOs on top of this model. In addition, we addressed the question of whether

★ ★ [32]Let us recall that this refers to the categorical model (first introduced in Chapter 2 and commented in Subsection 0.3.4) and its mapping to the underlying logical layer (see Subsection 3.3.1).

it is more user-friendly (1) to take a minimalist approach, i.e., to define only basic operations from which the user composes complex operations (e.g., see [19]), or (2) to add complex but frequently called operations to the basic set of SMOs (e.g., see [116]). We have opted for a combination of approaches, whereas SMOs[33] can be classified into three tiers:

1. *Basic operations.* Arbitrary modification of the unifying conceptual model can be achieved by any of the six basic operations, i.e., addObject, deleteObject, addMorphism, deleteMorphism, addId, dropId, or their combination. Nevertheless, although these are expressive operations, they are not very user friendly.

2. *Complex operations.* To enhance user experience, we introduce a set of complex operations, i.e., addRelationship, addList, addSet, addMap, addHierarchy, addStructure, addProperty, addRef, renameProperty, deleteRelationship, deleteList, deleteSet, deleteMap, deleteHierarchy, deleteStructure, deleteProperty, and dropRef. These operations internally combine basic operations (and implement this composition efficiently) and allow for modifications to the conceptual model in a user-friendly way.

3. *Frequently called operations.* Based on typical user requirements, we introduce a group of frequently called operations, i.e., copy, move, group, ungroup, union, split, and changeCardinality. These operations are internally composed of basic and/or complex operations.

We provide the most user-friendly SMOs in the Table 0.6. Note that our approach is the only one that allows SMOs in the (logical) model-level (challenge E5), i.e., we explicitly consider cross-model redundancy. Moreover, as a result of the unification of the underlying model constructs, the SMOs we propose are not platform dependent, i.e., there is no need to distinguish between, e.g., embedding or referencing – that is a logical layer detail (challenge E2).

The proposed SMOs are declared in the platform-independent Multi-Model Schema Evolution Language (MMSEL) (see Chapter 5). An obvious part of the language is the ability to select a subset of records of a given kind over which to execute SMOs (i.e., a selection). Analogous to existing solutions, MMSEL expressions are translated into domain specific language (DSL) utilising so-called wrappers. Note that we also exploit a mapping between the conceptual and logical layers for translation into DSL. Compared to existing solutions for NoSQL and multi-model systems, our approach also propagates SMO to identifiers and references (challenge E4). ★

SMOs are propagated into the data using the following data transformation algorithms:

- The *Model-to-Category transformation* (see Subsection 3.4.1) is applicable for data translation between logical and categorical layers. The algorithm first reads the input data from the logical model and inserts the records into a unifying data structure that structurally corresponds to the schema category (i.e., the unifying data structure). ★

[33] These operations are discussed in more detail in the journal article "A Unified Evolution Management of Multi-Model Data Using Category Theory", which is currently unfinished, hence unpublished. The expected completion is Q3 2022.

★ - The *Category-to-Model transformation* (see Subsection 3.4.2) converts the data from a categorical to a logical representation in three steps:

    1. The *DDL algorithm* allows the translation of a unified schema into a platform-specific schema (i.e., it creates, e.g., statements CREATE KIND, ALTER KIND).
    2. The *DML algorithm* creates a list of DML statements to store the data in the logical layer (e.g., INSERT INTO KIND).
    3. Finally, the *IC algorithm* ensures specification of identifiers and references at the logical layer (e.g., ALTER KIND).

The main contribution of the algorithms is their universality for migration or changing the logical representation of data. Moreover, the input and output of data migration algorithms may be the data represented by a combination of logical models. Hence, we have multi-model to multi-model data migration (for ★ more details see Chapter 3). SMOs are propagated into the data by the eager strategy. Extending the support to other data migration strategies, e.g., lazy and hybrid strategies [110], forms our current and future work. In addition, we add the possibility of scheduling data migration with respect to the usual database system workload, i.e., data will be migrated not only when needed (i.e., lazy migration), but also proactively during lower workloads.

Furthermore, SMOs are classified into heavy and light operations. *Heavy operations* are based on the traditional concept, i.e., they are propagated to the data and to the mapping between the conceptual and logical layers, thereby increasing the workload on the side of the database system. The examples of heavy operations include delete, move, union, and split. On the contrary, *light operations* propagate changes only to the mapping, i.e., no immediate propagation to the logical representation of the data is required (challenge E10). Examples of light operations includes add, rename, and group. Note that some heavy ★ operations can be light under certain conditions and vice versa (see Chapter 5).

Categorically speaking, schema modification operations (SMOs) correspond to functors, i.e., structure preserving mappings between two (schema) categories (inspired by the approach [20]). Regarding data migration, we propagate SMOs that add new schema elements using pushouts (inspired by the approach [19]) and changes that only duplicate or add no new elements are propagated using a pullback functor and its right and left adjuncts (inspired by the approach [20]).

Finally, the proposed approach was verified in the academic prototype *MM-evocat* [5],[34] which currently supports evolution management and backwards propagation for data stored in *PostgreSQL* (a representative of a multi-model DBMS), *MongoDB* (a representative of a document DBMS), and *Neo4j* (a representative of a graph DBMS). The advanced evolution management tasks, e.g., propagating schema changes to queries (challenge E3), extracting changes from schema-less data and from queries (challenge E6), proposing an approach for cost estimation of data migration operations (challenge E8), and involving artificial intelligence in the schema and data evolution process (challenge E11), constitute our current and near-future work.

[34] https://www.ksi.mff.cuni.cz/~koupil/mm-evocat/index.html

Paper I

Categorical Management of Multi-Model Data

Irena Holubová[@1], Pavel Čontoš (Koupil)[1], Martin Svoboda[1]

Published in 25th International Database Engineering & Applications Symposium (IDEAS 2021) by Association for Computing Machinery
doi: 10.1145/3472163.3472166

[@] corresponding author, e-mail: irena.holubova@matfyz.cuni.cz

[1] Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Abstract

In this vision paper, we introduce an idea of a framework that would enable us to model, represent, and manage multi-model data in a unified and abstract way. Its core idea exploits constructs provided by category theory, which is sufficiently general but still simple enough to cover any of the logical data models used in contemporary databases. Focusing on promising features and taking into account mature and verified principles, we overview the key parts of the framework and outline open questions and research directions that need to be further investigated. The ultimate objective is to pursue the idea of a self-tuning system that would permit us to collapse the traditionally understood conceptual and logical layers into just a single model allowing for unified handling of schemas, data instances, as well as queries.

Keywords

- Multi-model data • Category theory • Data modeling

1.1 Introduction

The *variety* feature of Big Data inciting the so-called *multi-model data* has opened a challenging direction of data management. The (primarily) academia-driven approach, represented mainly by *polystores* [8], is based on the idea of *polyglot persistence*, i.e., the usage of a *mediator* managing 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* [13] that offer the support of multiple models under the hood of a single system, treating all the data models as first-class citizens [12]. Both the approaches have to face the same challenges brought by contradictory features of different but interlinked models and their data management specifics. For example, there are structured/semi-structured/unstructured formats, systems based on strong/eventual consistency, schema-full/schema-less/schema-mixed systems, declarative/functional query languages, etc.

In this vision paper, we describe the necessary steps that need to be carried out in order to provide a full-fledged and universally applicable solution to the problem of multi-model data management. We argue that a highly promising approach could be based on *category theory* [133], a theory general enough to cover all the currently popular data models (and probably even more) and having a sound mathematical background important for the efficient and correct data processing. Exploiting the constructs it offers, we provide a vision of a complex framework that would allow us to merge the conceptual and logical layers into just a single model through which schema descriptions, data instances, as well as query expressions could be handled in an entirely uniform and abstract way.

Based on the inspirational related work and with the help of a set of illustrating examples, the main contributions of the paper are: (1) discussion of ways how category theory can bring answers and solutions to the aspects of schema modelling, data representation, querying as well as evolution, transformations or

migration, (2) identification of the related open questions and research directions that need to be further investigated so that the framework can be applied in polystore and multi-model scenarios, and (3) the actual description of the envisioned category-based unified autonomous DBMS, including its main advantages with respect to the contemporary systems.

Section 1.2 provides a brief introduction to category theory, Section 1.3 then thoroughly introduces the key parts of the framework, so that in Section 1.4, we summarise advantages of the category-based unified DBMS and conclude.

1.2 Categories

Formally, a category $\mathbf{C} = (\mathcal{O}, \mathcal{M}, \circ)$ consists of a set of objects $\mathcal{O}$ serving as graph vertices, a set of morphisms $\mathcal{M}$ acting as directed edges, and a composition operation $\circ$ for the morphisms.

Each morphism is modelled and depicted as an *arrow* $f : A \rightarrow B$, where $A, B \in \mathcal{O}$, A being referenced to as a *domain* and B as a *codomain*, respectively. Whenever $f, g \in \mathcal{M}$ are two morphisms $f : A \rightarrow B$ and $g : B \rightarrow C$, it must hold that $g \circ f \in \mathcal{M}$, i.e., morphisms can be composed using the $\circ$ operation and the composite $g \circ f$ must also be a morphism of the category (i.e., transitivity is required). Moreover, $\circ$ must be associative, i.e., $h \circ (g \circ f) = (h \circ g) \circ f$ for any morphisms $f, g, h \in \mathcal{M}$, $f : A \rightarrow B, g : B \rightarrow C$, and $h : C \rightarrow D$. 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 $f : A \rightarrow B$ (obviously serving as a unit with respect to the composition).

As an example, let us mention at least the **Set** category, where objects are arbitrary sets and morphisms functions between them. Note, however, that both objects and morphisms can, in general, represent abstract entities of any kind.

Considering the related work, category theory is a promising solution for the indicated issues. Existing *bottom-up* approaches start from a single logical model (relational [134, 20], object-relational [19], or CSV/document/RDF [135]) and define a respective schema category and operations using standard categorical approaches (such as functors). A *top-down* approach from [18] defines a schema category covering various conceptual modelling approaches, but only with respect to the most common model of that time – the relational model. In this paper, we also follow this direction, however, with respect to multiple interlinked data models at a time. In other words, we will show that the categorical approach can also be used for the multi-model world.

1.3 Framework

The objective of the following text is to describe features, requirements, and principles of the core components of the envisioned database framework that would allow us to formally grasp the existing scenarios of polystores and multi-model databases, as well as could potentially contribute to the proposal of a truly unified and conceptual database system.

1.3.1 Multi-Model Scenario

In order to demonstrate the intended functionality, let us assume a sample multi-model scenario we will use throughout the individual illustrative examples.

Example 1.1. Figure 1.1 provides an example of a multi-model scenario. The relational model (violet) contains general information about customers, whereas the graph model (blue) captures their mutual friendship. The document model (green) maintains orders which are bound with particular customers using the wide-column model (red). □

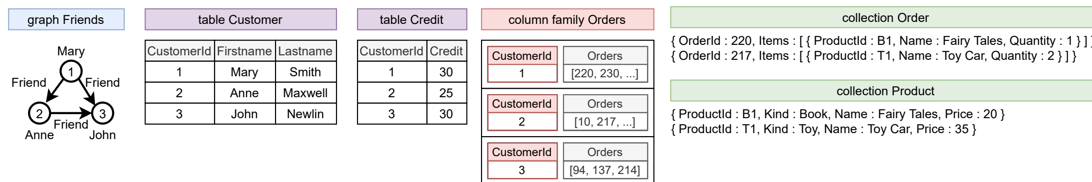


Figure 1.1: Sample multi-model scenario

The range of the available and widely used logical models is, of course, wider than the particular models we incorporated into our sample scenario. While the traditional relational model was and still is used as the primary option, a recently emerged family of NoSQL systems enabled wider usage of key/value, wide column, document, and graph models, too. As we have mentioned, their contradictory features, unfortunately, increase the complexity of handling the multi-model data in a truly universal way.

For the purpose of describing schemas of data at the conceptual layer (which intentionally abstracts and conceals specific details of the underlying logical models), existing mature and frequently used modelling languages such as ER [31, 136] and UML [32] (class diagrams in particular) can be exploited (see Figure 1.2 for the sample scenario). Unfortunately, while ER is more expressive and suitable for describing complex real-world relationships, it is not well-formalised and exists in various notations differing not just visually. On the other hand, UML is standardised, but only too data-oriented (lacking, e.g., weak entity types or other constructs).

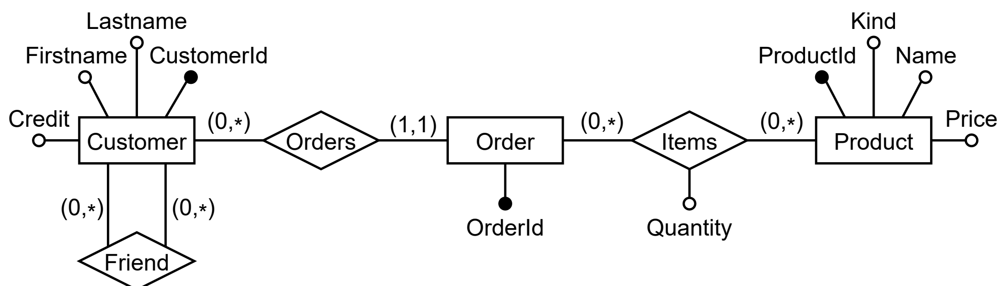


Figure 1.2: ER schema diagram for the sample data

Although this probably should not be the case, the main disadvantage of both the existing conceptual approaches is that they are actually not entirely suitable for grasping distinct data structures assumed by different logical models, simply because they are actually very closely related and inspired by the traditional

relational model. This means they are built on top of the notion of ordinary sets of tuples in the first normal form, and so clearly not fully conforming to the principles and nature of the other models that permit, e.g., repeated values, union types, or hierarchically constructed properties.

There actually already exist papers dealing with ER modelling of multi-model data (e.g. [137, 138]), primarily for polystores. Unfortunately, they lack crucial details of mapping from the conceptual layer to the particular DBMSs, which strongly influences inter-model references, model overlapping, query optimisation, evolution management, etc.

1.3.2 Schema Modeling

As a consequence of the previous observations, a new strategy capable of unified multi-model schema description and data representation needs to be found. We believe that an approach based on category theory could be promising enough, not just because of its universality and formal theoretical background, but also because of the already outlined existing approaches, though they are suffering from various drawbacks and are not yet ready for the multi-model scenarios.

The following idea of a *schema category*[1] can form the basis for the categorical conceptual modelling via which we will be able to describe the intended structure of the data together with basic integrity constraints. Borrowing the terminology from ER, objects of this category will represent individual entity types, attributes, as well as relationship types (without necessarily needing to distinguish between them later on, in the optimal case). In order not to disallow entity types with several or composed identifiers, super-identifiers can be exploited to encompass all of them, if needed. Morphisms will then interconnect the corresponding objects, allowing us to model the traditional concept of relationship cardinalities and attribute multiplicities through (min, max) constraints, where $min \in \{\mathtt{0}, \mathtt{1}\}$ and $max \in \{\mathtt{1}, *\}$. Obviously, $min \in \{\mathtt{0}, \mathtt{1}\}$ would restrain the lower bound of a number of occurrences (optional vs. compulsory) and $max \in \{\mathtt{1}, *\}$ the upper bound (at most one vs. at least one).

In order to make our category visualisations easier, we follow the convention that identity and non-core morphisms belonging to the transitive closure over the composition operation are entirely omitted, and that morphisms are labelled only with cardinalities different from $(\mathtt{1}, \mathtt{1})$, being treated as the default.

Example 1.2. In Figure 1.3, we can see the schema category corresponding to the ER model from Figure 1.2. For instance, the object (node) *Customer* represents the entity type *Customer*. Neighbouring objects connected via morphism (edges) correspond to its attributes (*Firstname*, *Lastname*, etc.) as well as relationship types (*Orders* and *Friend*) with respective cardinalities. □

Even though schema categories could also be designed directly by database users, this strategy might not be considered convenient enough, not just because of technical aspects that need to be carefully followed and ensured. Thus the outlined concept should primarily be understood as a means for internal database

[1] For the sake of simplicity, we omit technical details and more complex constructs. A separate paper [2] is devoted to the proposal of the schema category and the transformation process. In addition, the category itself can be designed variously, which is one of the open problems of the idea.

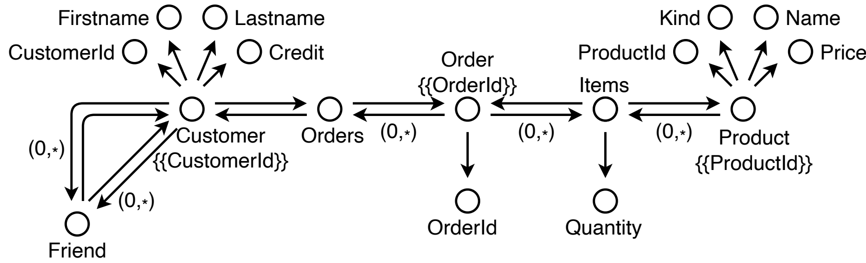


Figure 1.3: Schema category for the sample data

schema representation rather than a modelling language as such. This, in other words, means that a corresponding user-friendly modelling language would need to be proposed, too.

Moreover, in order to enable the option of deploying the envisioned framework in the context of the existing systems, transformations of ER and UML schemas, as well as at least semi-automatic inference methods capable of deriving schema categories from sample input data, would need to be proposed, too. More broadly, principles of generic schema management should be followed, similarly as in [139].

1.3.3 Database Decomposition

The very idea behind polystore and multi-model database scenarios is that different parts of the data are logically modelled, physically stored, and further processed differently by means of the corresponding logical models and query languages they are accompanied with. Having the schema category, the next step is to decompose it into such parts, let us call them *database components*. In practical terms, each of these components is expected to correspond to one of the underlying systems involved in a polystore (each one of them is accessible and queryable, yet specifically), and logical models involved in a multi-model database (with data directly and natively retrievable), respectively.

Each of these components would consist of a set of particular selected objects and morphisms, as if forming kind of a subgraph of the entire category that permits to incorporate the individual morphisms even without their ending objects (domain and codomain). Moreover, these components might be and most likely will often tend to intentionally be disconnected and/or more-or-less overlapping in real-world use cases. While the former aspect would require to take into account also the derived morphisms during the decomposition process, i.e., non-core morphisms derived using the category composition operation, the latter one finds its applicability in deliberate data redundancy across logical models, and so potentially improving query evaluation efficiency for data that is expected to be accessed together.

Example 1.3. Sample schema category decomposition into particular data models (depicted using the colours from Figure 1.1) is visualised in Figure 1.4. For instance, the document (green) component covers orders. The graph (blue) and relational (violet) components partially overlap, since we maintain the respective attributes (*Firstname* and *CustomerId*) in both the models. □

Focusing on a particular component, its objects and morphisms will then need to be internally mapped to particular logical constructs and structures provided

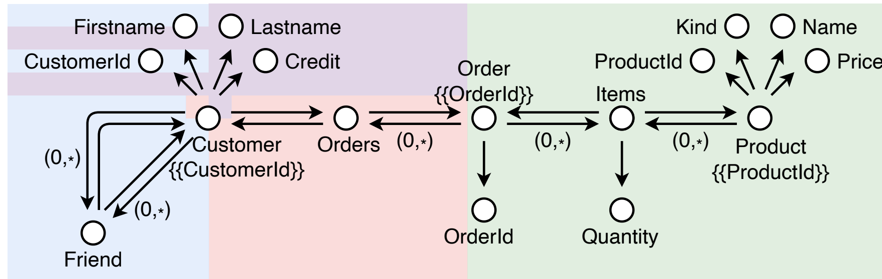


Figure 1.4: Decomposition into database components

by a given model, e.g., tables (together with their columns, primary keys, foreign keys, or other constraints) in case of the relational model, or collections of JSON documents (and their objects, arrays, and values) in case of the document model assumed by MongoDB.

Example 1.4. Let us consider the document (green) component in Figure 1.4. There exist several ways how to map this part of the category to one or more documents. Using a (semi-)automatic approach, the developer may choose, e.g., the following JSON schema (expressed symbolically):

```
{ OrderId, Items:  [ { ProductId, Name, Quantity }* ] }
{ ProductId, Kind, Name, Price }
```
□

1.3.4 Data Representation

Data assigned to each of the outlined database components is stored and logically represented differently, thus we have to find a way how such diversity could be encompassed within just one kind of a unifying data structure that would be capable of handling all the specifics of the individual models, but would still be simple enough. It is also worth realising that this structure should serve not just as a means of modelling the data actually present in the database (and so conforming to its conceptual schema), but also a means to represent results of the evaluated query expressions, including the intermediate ones.

While the existing multi-model DBMSs [13] more or less painfully create an extension of the data structures used for the original single model, there exist proposals of more general approaches, too. E.g., the *NoSQL Abstract Model* (*NoAM*) [27] represents the data as named collections, each containing a set of blocks consisting of a non-empty set of entries. *Associative arrays* [28] are then defined as mappings from pairs of unique (column and row) keys to values. Or the *Tensor Data Model* (*TDM*) [29], which introduces the idea of generalised matrices. We believe, however, we can go significantly further.

It would only be beneficial if the desired data structure could be derived from the outlined schema category. Such an objective could be achievable through an *instance category*. The idea is that this category would have exactly the same objects and morphisms when compared to the corresponding schema category, they would only differ in what the objects and morphisms mean and contain, i.e., what they are supposed to internally model. Perhaps tables, i.e., sets of tuples, could be utilised as a basis, at least for now. Hence, the idea is that each entity/relationship/attribute object could contain a set of values belonging to the active domain, and morphisms mappings of pairs of such values.

Example 1.5. Figure 1.5 provides a part of the instance category bearing the particular information about each product, i.e. *ProductId* and *Name*. □

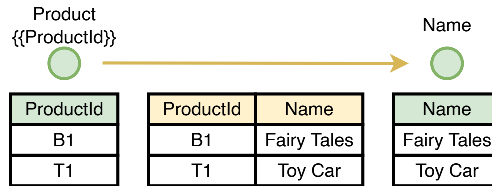


Figure 1.5: Part of an instance category

Obviously, it may not have been a good direction to stick with the principles of the relational model since complications at least with the ordering of values, duplicates, or non-trivial cardinalities of sub-attributes will unavoidably appear. Nonetheless, having the instance category proposed, a particular data format and techniques for data transformations will then also be needed (analogously to [134, 56]) so that a unified instance category can be obtained for input data in a particular format/model as well as such a category serialised in the opposite direction. Such low-level transformations will then find their essential position in data migration, evolution, or database self-tuning processes.

1.3.5 Query Language

The core functionality of each database system lies in its capability to query the data stored within it [140]. For obvious reasons, the expressive power of the provided query language must be sufficient with respect to the intended purpose and usage [141], though it may significantly differ across the models, systems, and particular languages, too. Similarly, evaluation of query expressions as such must be efficient enough. The major limiting disadvantage of the existing settings is that users must be thoroughly aware of the logical schema of the data. It means that the languages and so the query expressions are, not surprisingly, tightly bound with the structure of the data to be queried. While this can be acceptable in single-model systems, it no longer is in truly multi-model ones.

Therefore we advocate for finding ways of querying that would genuinely be conceptual, though expressive enough from the practical point of view. In other words, the goal is to find a unified conceptual query language that would permit to query the data without any further knowledge of the database decomposition (possibly even changing through time) and regardless of the specifics of the individual involved logical models. This means one query language, one query constructs, one syntax. Moreover, it should also be closed with respect to the data model, i.e., that both extensional data as well as intermediate and final query results are modelled uniformly, via instance categories.

The envisioned querying could be based on sub-graph pattern matching widely exploited in graph databases, simply because graph models are broadly understood as the most complicated ones, as well as because categories in general are tightly related to graphs. The idea is to describe one graph pattern we are looking for in terms of a *pattern category*, consisting of only the selected objects and morphisms available in the corresponding schema category. The individual objects

could be associated with selection conditions acting as domain filters. These conditions may be however complicated (with ranges, enumerations, conjunctions, disjunctions, ...), but may only involve a given object. Similarly, even morphisms could be equipped with conditions (comparisons, ...), either filtering or joining, always referencing only the pair of involved objects.

Example 1.6. Figure 1.6 introduces a pattern category for a query that should return names, kinds, and prices of all books or toys which can be bought by a customer with name Mary. As we can see, the black parts correspond to the schema category objects and morphisms, and so the database contents, whereas the red part is a newly introduced morphism enabling to join the two fragments using a joining condition. Grey-filled circles represent projection (objects that should appear in the final result). □

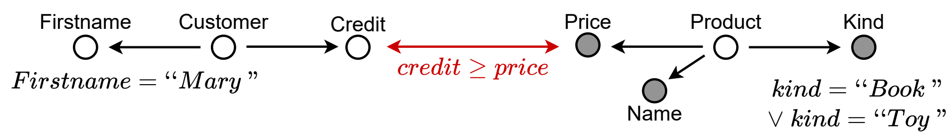


Figure 1.6: Pattern category for a sample query

Morphisms that were taken over from the schema category basically act as the corresponding inner joins, extensionally defined directly by the contents of the database. If it happens there are two or more disconnected parts (only assuming the adopted morphisms), they are joined using the Cartesian product (cross join), or a theta-join in case there are newly added morphisms with further joining conditions. It is also important to realise that particular schema objects and morphisms can be reused repeatedly in a query pattern. All in all, everything described inside one pattern category is expected to be satisfied as if in conjunction, i.e., based on the all-or-nothing principle.

The notion of a pattern category obviously cannot be used in order to describe more complicated queries. Therefore the next natural step is to allow ourselves to construct a *query category* through which we would be able to combine several simple pattern categories by advanced query constructs and operations modelled as special query objects. That would hopefully permit to incorporate not just the following widely considered constructs: set operations over the patterns (union, intersection, difference), existential and general quantifiers including their negations, or optional patterns. Though there will undoubtedly be a variety of not just technical obstacles, we also need to be able to cover disjunctions of patterns through which complicated disjunctive conditions involving several objects could be expressed. Similarly, non-binary filtering or joining conditions, grouping and aggregation, or derivation of new values not present in the database via arithmetic expressions, function calls, etc.

Syntax of the actual query language as such will also need to be carefully designed so that it permits to express all the ideas but still remains user-friendly enough. While for the purpose of describing the individual pattern categories the idea of ASCII-art-inspired syntax assumed by Cypher query language in Neo4j could most likely be exploited, principles of the composition of more complex queries is a more significant challenge, possibly benefiting from the ideas

of sub-queries or chaining of clauses, or simply the existing proposals such as SQL++ [142].

1.3.6 Query Evaluation

Based on the knowledge of the schema category, database decomposition, and internal schema mappings, the query category now needs to be decomposed into individual *query parts*, each to be evaluated separately so that it produces the corresponding intermediate result modelled in terms of instance categories with their schemas. Each query part consists of the selected objects and morphisms from the query category, yet not all of them do necessarily need to be involved. In fact, in the case of cross-model queries, evaluation of certain morphisms and/or query objects will intentionally need to be postponed, simply because no database component is capable of their evaluation on its own.

While in a polystore scenario each query part needs to be translated into the corresponding query expression within the specific query language a given system supports, and this expression internally evaluated so that the yielded result can be transformed into our unified logical representation, the corresponding intermediate result can directly be obtained from the database data files (at the physical layer) in case of a multi-model scenario.

Example 1.7. Our decomposed inter-model query is depicted in Figure 1.7. The relational part (violet) extracts credits of customers named Mary:

```
SELECT T2.Credit
FROM Customer AS T1 NATURAL JOIN Credit AS T2
WHERE T1.Firstname = "Mary";
```

Analogously, using the MongoDB query language, the document part (green) extracts all properties but ProductId of books or toys:

```
db.Product.find(
   { Kind: { $in: [ "Book", "Toy" ] } }, { ProductId: 0 }
);
```

□

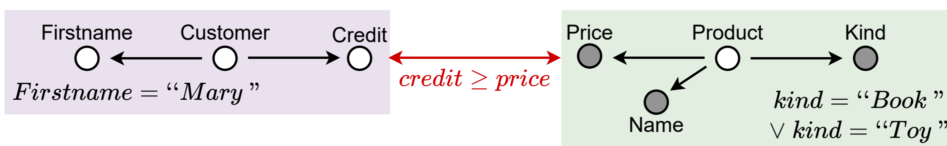


Figure 1.7: Query category decomposition

Note that multiple query parts belonging to the same database component may be generated, because the expressive power of a given query language may not be sufficiently high. For example, we can expect separate query parts and so multiple *find* queries for each of the involved collections in the case of MongoDB. Similar issues can appear in key/value or wide column databases, too.

Last but not least, while the query translation process takes a query category part and rewrites it to a specific query expression, we also need to deal with the opposite direction so that the existing query expressions in various at least widely used languages can be taken and migrated to our categorical representation.

Having a given query part evaluated, the obtained intermediate result needs to be represented as an instance category with a structure conforming to a newly

defined schema category. For example, having an intermediate result in a form of a table retrieved from a relational system or a collection of possibly projected documents from MongoDB, its corresponding schema category could be constructed by *contracting* the given query category part (its objects and morphisms) into a star with all the involved attribute objects (simple or with sub-attributes) collected around the central object representing the super-identifier of the entire obtained result. The central object is derived from the *backbone* objects and morphisms of the query part, i.e., as a result of more-or-less complicated joining process.

The schema contraction step is essential in order to support advanced query constructs such as groupings, aggregations, or derivation of new values in general. As a consequence, however, we may lose the interpretability of certain objects or morphisms with respect to the original schema.

Once all the recognised query parts are evaluated and so the intermediate results obtained, the evaluation of all the postponed and not yet considered morphisms and/or query objects can be completed at the unified layer of the framework. This means that we need to evaluate these morphisms/objects one by one, partially contracting the corresponding parts of the current query category, step by step, so that we eventually retrieve the result of the entire query in the very end.

Example 1.8. As depicted in Figure 1.8, the intermediate results for the relational (violet) and document (green) parts of the query were transformed into instance categories with the respective contracted schemas. By evaluating the postponed theta join, we retrieved the overall query result. □

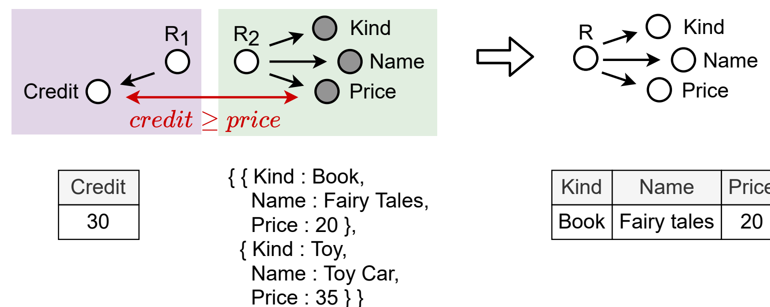


Figure 1.8: Intermediate and final query results

Evaluation and optimisation of queries are no doubt absolutely crucial aspects defining the actual practical usability of database systems. Thus multiple heuristically designed query evaluation plans need to be considered so that the one with the lowest cost can be selected and actually executed as the only one. Apparently, the efficient evaluation also needs to be supported by indices, in this case model-agnostics [12].

Example 1.9. An alternative (though probably less efficient) query plan depicted in Figure 1.9 involves, in addition, also the graph model (blue), where names of customers can also be extracted. □

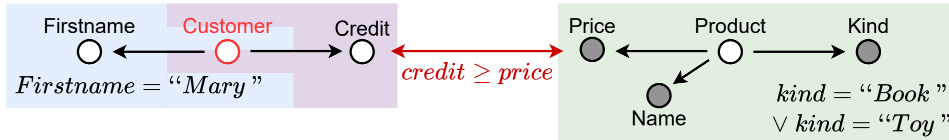


Figure 1.9: Alternative query evaluation plan

1.3.7 Evolution Management

Evolution management is a process of preserving the integrity of the whole system when user requirements change. A change in the structure of the data may require changes in related data instances, queries, integrity constraints, storage strategies, etc. Thanks to the general representation of all the data models using the schema category, we can define a unified cross-model set of schema modification operations (SMOs). The abstract categorical layer also enables backward compatibility of the queries even when the storage strategies and related mappings may change.

Example 1.10. An example of evolution might involve merging of attributes Firstname and Lastname into one attribute Fullname. In addition, the data about customers can be migrated from the graph model to the relational model only. In Figure 1.10, we can see the old and new versions of the affected parts of the schema category. While the query accessing the modified part must be adapted accordingly in the first case, the schema category and so the categorical queries remain the same in the second case. □

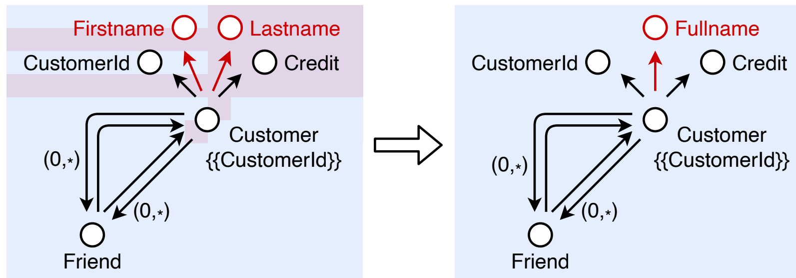


Figure 1.10: Old and new versions of schema category

The existing multi-model evolution management approaches provide an interesting inspiration, but they also have various significant limitations, such as, they only focus on aggregate-oriented NoSQL systems [122] or they omit critical inter-model links [117]. A sufficiently general set of SMOs can most likely be borrowed from [143].

1.4 Summary and Conclusion

The outlined framework has many features that seem to be promising and are undoubtedly worth further exploration. Either way, it can be deployed in both the polystore and multi-model scenarios, i.e., it is applicable to state-of-the-art systems. While we believe it can become a basis of newly designed database management systems, it still relies on, integrates, or extends various existing

verified approaches, as well as it further elaborates concepts already advocating such objectives [12].

In other words, we propose to go beyond the ideas and boundaries of the contemporary systems toward entirely unified and conceptual databases that would allow us to fully abandon the idea of various logical models together with their distinct data structures, specific features, and often proprietary query languages, as it is demonstrated in Figure 1.11. Though such systems could be based on category theory, as we have outlined, other well-established formalisms could possibly serve as well.

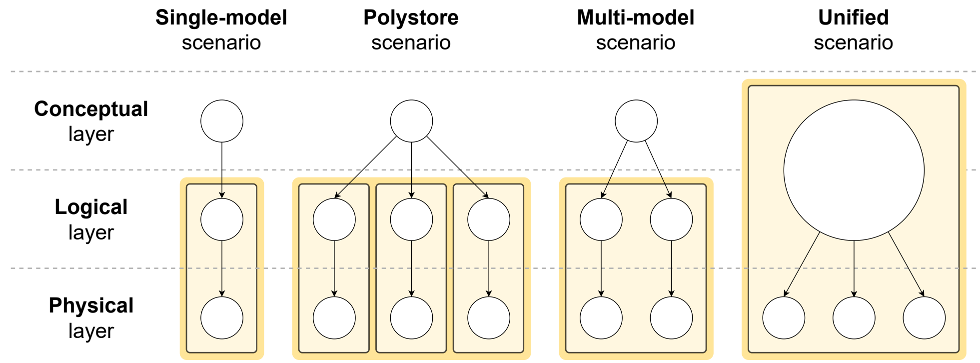


Figure 1.11: Layered architecture of database systems

It is apparent that there are only too many existing systems, models, formats, and languages. If it is difficult for the users to get sufficiently acquainted with such approaches and make decisions whether and when to use them in single-model situations, the more challenging the task it becomes when polyglot persistence is followed, i.e, when multiple models and systems should be considered, understood, deployed, and maintained at the same time. Obviously, at least from the long-term perspective, it is highly unlikely that such a variety together with currently ongoing trends could remain sufficiently sustainable.

Besides the already mentioned key aspects of the envisioned self-tunable database management system, we think the emphasis should be put on the formal background, unification, conceptuality, and user-friendliness. In other words, it is necessary to find a reasonable balance between the utilisation of high-level formal theories such as the one assumed, and the practical applicability of the whole approach on the other hand. The core differences, features, advantages, and contributions represented by the framework we envisioned are summarised as follows.

- Intended schema of the data is described only once and at the conceptual platform-independent layer, only targeting real-world entities and relationships they can enter into.
- Schema categories have higher expressive power than the traditional ER or UML languages, seamlessly allowing to work with structured attributes or other constructs.
- Attributes, entity types, and relationship types no longer need to be mutually distinguished and handled differently, and so the entire modelling process can be simplified.

- Instance categories are general enough to serve as a universal data structure for all the currently widely used models, permitting the integration of future suitable models, too.
- The involved logical models play mutually equal roles, none of them is given priority, as is often the case in the existing multi-model systems.
- The traditional conceptual and logical layers collapsed into just a single unified representation, so they no longer need to be considered separately.
- Handling of the data variety aspect is pushed from the former logical layer to the physical one, where various file organisations and indices can appropriately be utilised.
- Decomposition into different models became internal with no impact on the users, and so they do not need to be aware of it, nor are they forced to adjust the style of thinking with respect to the particular models involved.
- Individual components can intentionally overlap each other, as well as do not necessarily need to contain all the data so that long-running migrations can be supported.
- Though the users can supervise the decomposition process, it can be fully autonomous and capable of reorganising the data based on the changing workload or other aspects.
- Query language and its constructs are conceptual, and so independent on the internal representation of the data, not requiring the users to have its deeper knowledge.
- Querying itself is based on graph pattern matching, a mature enough principle from graph databases allowing for the evaluation of complex queries.
- The query model is closed with respect to the input data and intermediate/overall results, all consistently modelled in terms of instance categories.
- Schemas, data, as well as queries are all modelled via homogeneously designed categories, all following the same structure and principles.
- Inter-model references or links of other kinds and cross-model queries are supported natively, which is not the case of many an existing system.
- Category theory provides a strong formal background, which brings the potential for advanced query optimisation techniques that would otherwise be difficult to achieve.

Although we covered the key components concerning schema modelling, data representation, and querying, there are also other important aspects of multi-model data management we did not cover. We also did not want to provide a complex solution but to outline promising research directions and encourage both researchers and practitioners to pursue the envisioned ideas.

Paper II

Categorical Modeling of Multi-Model Data: One Model to Rule Them All

Martin Svoboda[@1], Pavel Čontoš (Koupil)[1], Irena Holubová[1]

Published in International Conference on Model and Data Engineering (MEDI 2021), Lecture Notes in Computer Science (vol. 12732) by Springer
doi: 10.1007/978-3-030-78428-7_15

[@] corresponding author, e-mail: martin.svoboda@matfyz.cuni.cz

[1] Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Abstract

Following the Gartner predictions, most of the DBMSs, both relational and NoSQL, have become multi-model. However, this functionality brought plenty of related issues. The core problem is how to design a multi-model application. The step from the conceptual layer to a set of distinct interlinked logical models is not straightforward.

In this paper, we propose an approach based on category theory, which provides a unified view of the data and a strong mathematical basis for their management. We propose a schema and instance categories covering popular models and we show how an ER model can be transformed to such a categorical layer. We also introduce the whole framework based on the categorical model and discuss open research issues.

Keywords

- Multi-model data • Category theory • Conceptual modelling

2.1 Introduction

Despite the *one size does not fit all* argument [144], most of the popular database management systems (DBMSs), being relational, NoSQL, or other, have followed the Gartner predictions [145] of support of multiple data models.

Example 2.1. Consider an example of a multi-model scenario in Fig. 2.1. Social network of customers is captured using a graph. Additional information, such as their credit limit, is recorded in a relational table. Orders submitted by customers are stored as JSON documents. A wide-column table maintains the history of all orders, and a key/value mapping maintains current shopping carts. A cross-model query might return *friends of customers who ordered any item with a price higher than 180.* □

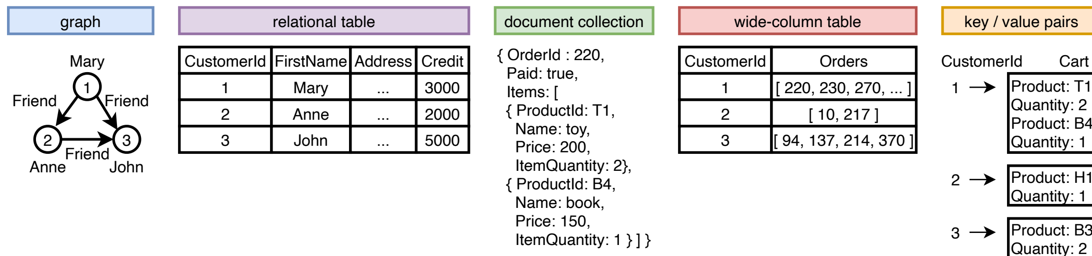


Figure 2.1: A multi-model scenario (partially borrowed from [11])

Though simple and natural in the idea – adding another data model – its realisation is quite complex, as the combined models (and respective systems) often have contradictory features. There are structured, semi-structured, and unstructured formats; systems based on strong or eventual consistency; declarative and functional query languages; etc. Currently, there exist more than 20 representatives of *multi-model databases* [146], involving well-known traditional relational and also novel NoSQL tools. Regarding the multiple models, they differ in the

following three core aspects: (1) the original model (relational, graph, document, etc.), (2) the strategy for its extension (new storage structures, new interfaces, etc.), and (3) the set of supported models.

Such a situation is difficult for users who want to develop a multi-model database application. According to the traditional recommendations, they would first create a conceptual schema (e.g., using ER or UML). Then, there exist verified recommendations on how to transform such a schema into, e.g., the relational model, whereas the existing relational DBMSs support this model more-or-less according to its well-defined standard. However, (not only) with regards to the above-described variations, the step from an ER/UML conceptual model to virtually any possible (not standardised) combination of multiple models is not that straightforward.

For this purpose, we need a 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; (3) perform correct and complete evolution management, i.e., propagation of changes; (4) enable data migration without complex reorganisations; and (5) permit integration of new data models. Although both ER and UML are strong enough to cover some of these points, their main purpose is different and not so wide. Hence, we argue that a new layer between the abstract conceptual model and particular logical models needs to be established to pursue the multi-model data management.

In this paper, we propose a solution based on *category theory* [133], a theory sufficiently general for the multi-model situation and providing a strong mathematical background for further management of the data, such as transformations between the models, cross-model querying, multi-model evolution management, etc. The main contributions of the paper can be summed up as follows:

- We define schema and instance categories covering all known data models.
- We provide the ER-to-category transformation algorithm.
- We introduce the whole framework based on the categorical model.
- We discuss a range of challenges opening an interesting research area.

Paper outline: In Section 2.2, we overview the related work and in Section 2.3 basics of category theory. We describe the ER-to-category transformation in Section 2.4 and the whole framework in Section 2.5. In Section 2.6, we discuss its general benefits as well as issues and open challenges. Finally, we conclude in Section 2.7.

2.2 Related Work

While the multi-model DBMSs [13] (more-or-less painfully) provide 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* (*NoAM*) [27] represents the data as named collections, each containing a set of blocks consisting of a non-empty set of entries. *Associative arrays* [28] are defined as mappings from pairs of unique (column and row) keys to values. Finally, *Tensor Data Model* (*TDM*) [29] introduces the idea of generalised matrices.

For the higher levels of abstraction, there also exist a few proposals. *TyphonML* [137] 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 [138], 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.

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 [134, 20], or object-relational, i.e., hierarchies of classes [19]) and define a respective schema category and operations using standard categorical approaches (such as functors). Paper [135] proposes a categorical approach for the relational (CSV), document, and graph (RDF) models, but only with intra-model data migrations and querying.

A *top-down* approach from [18] defines a schema category covering various conceptual modelling approaches, but unfortunately only with respect to the most common model of that time – relational. In this paper, we also follow this direction, however, with respect to multiple interlinked popular data models.

2.3 Preliminary Concepts

We first provide essential definitions related to category theory we need in this paper. Assuming a general knowledge of the ER model, we also describe our basic assumptions on ER conceptual modelling.

2.3.1 Category Theory

Formally, a category $\mathbf{C} = (\mathcal{O}, \mathcal{M}, \circ)$ consists of a set of objects $\mathcal{O}$ serving as graph vertices, a set of morphisms $\mathcal{M}$ acting as directed edges, and a composition operation $\circ$ for the morphisms.

Each morphism is modelled and depicted as an *arrow* $f : A \to B$, where $A, B \in \mathcal{O}$, A being referenced to as a *domain* and B as a *codomain*, respectively. Whenever $f, g \in \mathcal{M}$ are two morphisms $f : A \to B$ and $g : B \to C$, it must hold that $g \circ f \in \mathcal{M}$, i.e., morphisms can be composed using the $\circ$ operation and the composite $g \circ f$ must also be a morphism of the category (i.e., transitivity is required). Moreover, $\circ$ must be associative, i.e., $h \circ (g \circ f) = (h \circ g) \circ f$ for any morphisms $f, g, h \in \mathcal{M}$, $f : A \to B, g : B \to C$, 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 $f : A \to B$ (obviously serving as a unit with respect to the composition).

Example 2.2. **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 $(a, c) \in g \circ f$ whenever there exists at least one value $b \in B$ such that $(a, b) \in f$ and $(b, c) \in g$. □

Even though objects in real-world categories usually tend to be sets of certain items and morphisms functions between them, both objects and morphisms may actually represent abstract entities of any kind.

2.3.2 Conceptual Modeling

Although ER is well-known, there exists a variety of its particular forms and notations. Therefore we summarise the particular ER version [136] we assume in this paper. We suppose that each entity type must have at least one attribute, as well as it must always have at least one identifier, regardless it is strong or weak. As for multiplicities of attributes and cardinalities of relationship types, (min, max) ranges are assumed, permitting `0` or `1` in the lower boundary (expressing whether at least one occurrence is required or no) and `1` or $*$ in the upper (whether just at most one or even more occurrences are allowed). We also consider structured attributes, yet only flat, i.e., with one level of sub-attributes, and these must be bound to the base one by the default multiplicity $(\mathtt{1}, \mathtt{1})$. As for relationship types, they must not have their own identifiers. Finally, we also take into account ISA hierarchies and weak entity types.

2.4 Schema Representation

Our primary objective is a proposal of a conceptual schema representation based on categories that would allow us to provide a unified view of multi-model data. Although users might create such representations directly when modelling the data, the resulting categories will most likely be too complicated. Thus we primarily understand our solution as a means for schema and data representation.

Following a top-down approach, we describe how to translate a given ER schema into an equivalent categorical representation. In particular, two categories will be derived. The aim of a *schema category* **S** is to grasp the intended structure of the data, including basic integrity constraints. An *instance category* **I** then represents a particular database state, i.e., a database instance filled with data conforming to a schema defined by **S**. Both **S** and **I** are identical as for their structure, they differ in the internal contents of their objects and morphisms.

2.4.1 Schema Category

Schema category is defined as $\mathbf{S} = (\mathcal{O}_\mathbf{S}, \mathcal{M}_\mathbf{S}, \circ_\mathbf{S})$. Objects in $\mathcal{O}_\mathbf{S}$ represent individual entity types, attributes, and relationship types we have in the input ER schema. One for each one of them. We will distinguish *entity*, *attribute*, and *relationship* objects, but solely for the purpose of an easier explanation. Similarly, we will distinguish *attribute*, *identifier*, *relationship*, and *hierarchy* morphisms.

Assuming that $\mathbb{A}$ is a set of names of attributes in the input ER schema (or simply the active domain for these names), $\mathbb{E}$ the domain of entity type names, and $\mathbb{R}$ the domain of relationship type names, we will now have a look at the internal structures behind the category objects and morphisms.

In particular, each object $o \in \mathcal{O}_\mathbf{S}$ will internally be modeled as a pair $(superid, ids)$: $superid \subseteq \mathbb{A}$ will be a *super-identifier* of a given object, i.e., a set of attributes using which instances of a given attribute, entity or relationship type can be uniquely identified; $ids \subseteq \mathcal{P}(superid)$ will be a set of individual particular *identifiers* a given object is associated with. It must hold that $superid \supseteq \bigcup_{id \in ids} id$, i.e., that *superid* consists of at least all the attributes involved in the individual identifiers. In case of entity objects and attribute objects, equality will actually

hold, and so *superid* will form the minimal super-identifier. In case of relationship objects, there can be additional attributes.

Example 2.3. Having an entity type `Person` with one simple identifier `PersonId`, one composed identifier consisting of two attributes `FirstName` and `LastName`, and additional attributes `Age` and `Address`, the corresponding entity object $\texttt{Person}_{\mathbf{S}}$ will be modelled as a pair $(superid, ids)$, where $superid = \{\texttt{PersonId}, \texttt{FirstName}, \texttt{LastName}\}$ and $ids = \{\{\texttt{PersonId}\}, \{\texttt{FirstName}, \texttt{LastName}\}\}$. □

Next, each morphism $m \in \mathcal{M}_{\mathbf{S}}$ will internally be a pair (min, max), allowing us to model the traditional concept of relationship cardinalities and attribute multiplicities through (min, max) constraints: $min \in \{\texttt{0}, \texttt{1}\}$ restrains the lower bound of a number of occurrences (`0` for an optional occurrence, `1` for a compulsory one), $max \in \{\texttt{1}, *\}$ the upper bound (`1` for at most one occurrence, $*$ for at least one), respectively.

For each object $o \in \mathcal{O}_{\mathbf{S}}$, its identity morphism is denoted and defined as $1_o = (\texttt{1}, \texttt{1})$. As for the composition operation $\circ_{\mathbf{S}}$, having two morphisms $m_1 = (min_1, max_1)$ and $m_2 = (min_2, max_2)$ such that $m_1, m_2 \in \mathcal{M}_{\mathbf{S}}$, their mutual composite is evaluated as $m_2 \circ_{\mathbf{S}} m_1 = (\min(min_1, min_2), \max(max_1, max_2))$, simply choosing the lowest of limits for the lower bound and the highest for the upper one. For example, having $m_1 = (\texttt{0}, \texttt{1})$ and $m_2 = (\texttt{1}, *)$, the composite will be $m_2 \circ_{\mathbf{S}} m_1 = (\texttt{0}, *)$.

2.4.2 Instance Categories

Having introduced the schema category, let us now focus on the structure of instance categories $\mathbf{I} = (\mathcal{O}_{\mathbf{I}}, \mathcal{M}_{\mathbf{I}}, \circ_{\mathbf{I}})$, each representing a particular data instance constructed for a schema $\mathbf{S} = (\mathcal{O}_{\mathbf{S}}, \mathcal{M}_{\mathbf{S}}, \circ_{\mathbf{S}})$. As already outlined, objects as well as morphisms of $\mathbf{I}$ will be identical to objects and morphism of $\mathbf{S}$ as for their mere existence, they will differ in structures they internally represent, though.

Let us have an instance object $o_{\mathbf{I}} \in \mathcal{O}_{\mathbf{I}}$ associated with the schema object $o_{\mathbf{S}} = (superid, ids) \in \mathcal{O}_{\mathbf{S}}$ as its corresponding counterpart. The purpose of $o_{\mathbf{I}}$ is then to represent particular data instances conforming to $o_{\mathbf{S}}$. Assuming that $\mathbb{V}$ is the domain of all possible values of attributes, object $o_{\mathbf{I}} = \{t_1, t_2, \ldots, t_n\}$ for some $n \in \mathbb{N}_0$ will technically be modeled as a set of tuples, each of which will be represented as a function $t_i : superid \rightarrow \mathbb{V}$ for any $i \in \mathbb{N}, 0 < i \leq n$. This means that tuples themselves are mutually unordered, duplicate tuples are not permitted, and all attributes in a tuple must always be specified.

Example 2.4. Having an entity schema object $\texttt{Person}_{\mathbf{S}} = (superid, ids)$ introduced in the previous Example 2.3, its instance object could, e.g., contain the following tuples:

$$\begin{aligned}\texttt{Person}_{\mathbf{I}} = \{&\{(\texttt{PersonId}, \texttt{1}), (\texttt{FirstName}, \texttt{Mary}), (\texttt{LastName}, \ldots)\},\\ &\{(\texttt{PersonId}, \texttt{2}), (\texttt{FirstName}, \texttt{Anne}), (\texttt{LastName}, \ldots)\},\\ &\{(\texttt{PersonId}, \texttt{3}), (\texttt{FirstName}, \texttt{John}), (\texttt{LastName}, \ldots)\}\}.\end{aligned}$$ □

Moreover, we require these tuples to satisfy the identification ability of all the involved identifiers. In other words, the data must conform to the uniqueness requirement imposed by these identifiers. For this purpose, let us first introduce a standard projection operation π. If $o = \{t_1, t_2, \ldots, t_n\}$ is a set of tuples over a set of attributes A, $\pi_{A'}(o)$ for some $A' \subseteq A$, $A' \neq \emptyset$ is defined as a restriction

of o to the attributes contained in A' only. I.e., $\pi_{A'}(o) = \{t' \,|\, t' = \pi_{A'}(t), t \in o\}$, where $\pi_{A'}(t) = \{(a, v) \,|\, (a, v) \in t, a \in A'\}$. Now, we can say that $\forall\, id \in ids$ for our object $o_{\mathbf{I}}$ conforming to $o_{\mathbf{S}} = (superid, ids)$ it must hold that $|o_{\mathbf{I}}| = |\pi_{id}(o_{\mathbf{I}})|$.

As for the morphisms, they act as binary relations (i.e., they abide by the principles of the **Rel** category). In other words, having a morphism $m_{\mathbf{I}} \in \mathcal{M}_{\mathbf{I}}$, $m_{\mathbf{I}} : o_1 \to o_2$ for some $o_1, o_2 \in \mathcal{O}_{\mathbf{I}}$, it must then hold that $m_{\mathbf{I}} \subseteq o_1 \times o_2$. Moreover, provided $m_{\mathbf{S}} = (min, max) \in \mathcal{M}_{\mathbf{S}}$ is the corresponding schema morphism, the actual cardinality condition must be satisfied. It means that $\forall\, t_1 \in o_1$ it must hold that $|\{t_2 \,|\, t_2 \in o_2, (t_1, t_2) \in m_{\mathbf{I}})\}| = c$ must be within the cardinality limits, i.e., $c \geq 1$ in case of $min = \mathtt{1}$ and, at the same time, $c \leq 1$ in case of $max = \mathtt{1}$. Note that the other situations $min = \mathtt{0}$ or $max = *$ do not need to be treated as they are satisfied implicitly.

Identity morphism 1_o for each object $o \in \mathcal{O}_{\mathbf{I}}$ is defined as a function (just a special case of a more generic relation) $1_o = \{(t, t) \,|\, t \in o\}$. Finally, the composition operation $\circ_{\mathbf{I}}$ corresponds to, unsurprisingly, the composition in **Rel**.

2.4.3 Schema Translation

In ER schemas, it is usually assumed that names of entity types and relationship types are distinct. Attribute names are expected to be unique within the context of an entity type or relationship type they belong to (including the impact of the inheritance in the case of ISA hierarchies). In order to make our formal descriptions less complicated, we assume that, without loss of generality, all the names are unique globally. I.e., that $\mathbb{A}$, $\mathbb{E}$, and $\mathbb{R}$ are mutually distinct. As a result, names of all category objects we are about to create will directly correspond to the original entity types, relationship types, or attributes, respectively.

As for the visualisation of our categories, the following conventions will be applied to simplify their diagrams. We will label all objects with their names and entity objects with their identifiers. As for morphisms, we will not include loops for identity morphisms. Similarly, we will only include morphisms we explicitly need to construct, silently assuming that morphisms belonging to the transitive closure over the composition operation, though not visualised, in fact, are included, too. Names of morphisms will be entirely omitted; cardinalities will only be included in case they are different from $(\mathtt{1}, \mathtt{1})$. Furthermore, since names will often need to be calculated dynamically, we suppose that `a` will stand for the actual value of a variable a, as well as that names cannot contain the $\cdot$ symbol because we want to use it as an auxiliary separator. For example, if $a = \mathtt{Customer}$ and $i = 1$, expression `a·i` shall be interpreted as `Customer·1`.

Starting with an empty schema category $\mathbf{S} = (\mathcal{O}_{\mathbf{S}}, \mathcal{M}_{\mathbf{S}}, \circ_{\mathbf{S}})$, we will now describe how the individual constructs appearing in ER schemas are transformed into the individual objects and morphisms of the category.

It is important to realise that there may exist dependencies between the individual entity types occurring in the input ER schema because of ISA hierarchies or weak identifiers. They must be transformed in an order that does not violate such dependencies. Hence, they need to be acyclic, i.e., such an order must exist. So, hierarchies are transformed by starting at their root nodes and proceeding toward their leaves, always processing a parent node before its children. Similarly, weak entity identifiers can be transformed only when all the entity types they

depend on are already resolved. Finally, once all the entity types are processed, relationship types can be processed as well.

Entity Types.

Let $e \in \mathbb{E}$ be an entity type and $a_1, \dots, a_m$ its attributes for some $m \in \mathbb{N}_0$. Let each of these attributes $a_i \in \mathbb{A}$ be associated with a particular multiplicity (min_i, max_i), equalling to the default $(\mathtt{1}, \mathtt{1})$ whenever not provided explicitly. In case a given attribute a_i is a structured attribute, let then $a_i^1, \dots, a_i^{k_i}$ be its individual sub-attributes for some $k_i \in \mathbb{N}$. Finally, let $id_1, \dots, id_n$ be all the (strong) identifiers of e for some $n \in \mathbb{N}_0$, each modelled as a set of at least one attribute it consists of. It must hold that $id_j \subseteq \{a_1, \dots, a_m\}$ and $|id_j| \geq 1$ for any $j \in \{1, \dots, n\}$, as well as that only attributes with the trivial multiplicity can be involved in the identifiers, i.e., that $\forall\, a_i \in \bigcup_{j=1}^{n} id_j$ the multiplicity of a_i must only be $(\mathtt{1}, \mathtt{1})$. Note that there may not be even a single identifier in case e is a descendant in a hierarchy and/or contains at least one weak identifier. Transformation of both these constructs will be discussed later, and so for now, the described entity type e and its attributes and identifiers will contribute to the resulting schema category as follows.

First, an *entity* object $\mathtt{e}$ is created for e such that $\mathtt{e} = (superid, ids)$. Assuming that $u(a_i) = \{a_i\}$ for any unstructured attribute a_i and $u(a_i) = \{a_i^1, \dots, a_i^{k_i}\}$ otherwise, we can write $ids = \{id'_1, \dots, id'_n\}$ with $id'_j = \bigcup_{a_i \in id_j} u(a_i)$ for each $j \in \{1, \dots, n\}$, and $superid = \bigcup_{id' \in ids} id'$. In other words, identifiers are preserved as they were provided, we just need to unfold the involved structured attributes to the corresponding sub-attributes they consist of.

Next, each attribute a_i with $i \in \{1, \dots, m\}$ is processed, one by one. In case a_i is an ordinary (unstructured) attribute, an *attribute* object $\mathtt{a_i} = (\{a_i\}, \{\{a_i\}\})$ is created. Otherwise, i.e., when a_i is a structured attribute, the resulting *attribute* object equals to $\mathtt{a_i} = (\{a_i^1, \dots, a_i^{k_i}\}, \{\{a_i^1, \dots, a_i^{k_i}\}\})$. For each of these sub-attributes a_i^j, $j \in \{1, \dots, k_i\}$, another *attribute* object $\mathtt{a_i^j} = (\{a_i^j\}, \{\{a_i^j\}\})$ is produced, together with an attribute morphism $\mathtt{a_i} \cdot \mathtt{a_i^j} = (\mathtt{1}, \mathtt{1}) : \mathtt{a_i} \to \mathtt{a_i^j}$ binding it with its base attribute a_i. Finally and under all circumstances, an attribute morphism $\mathtt{e} \cdot \mathtt{a_i} = (min_i, max_i) : \mathtt{e} \to \mathtt{a_i}$ is yielded, binding the whole attribute a_i with the entity type e as such.

Example 2.5. Entity type `Person` depicted in Fig. 2.2 is transformed to an entity object `Person` having $superid = \{\mathtt{PersonId}, \mathtt{FirstName}, \mathtt{LastName}\}$ and $ids = \{\{\mathtt{PersonId}\}, \{\mathtt{Firstname}, \mathtt{LastName}\}\}$. Its attributes are transformed to the respective attribute objects `Email`, `FirstName`, etc., together with the given cardinalities. In case of the structured attribute `Address`, there is an attribute object `Address`, as well as attribute objects `Street`, `City`, and `PostalCode` for its individual sub-attributes. □

ISA Hierarchies.

Let us have a descendant entity type c and its parent entity type p. Note that there can only be one parent for c, since multiple inheritance is not permitted. Let $\mathtt{c} = (superid_c, ids_c)$ be the corresponding already partially resolved entity object for c (based on the so far described rules, i.e., only concerning its locally defined

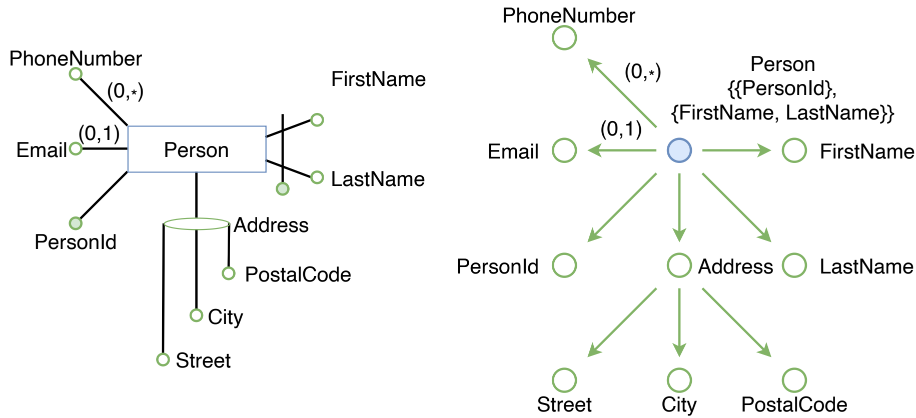


Figure 2.2: Translation of entity types and attributes

attributes and identifiers, if any), and $\mathtt{p} = (superid_p, ids_p)$ the fully resolved entity object for p (including the impact of inheritance and/or not yet described weak identifiers). We now describe how $\mathtt{c}$ will be altered to incorporate the effect of the hierarchy inheritance from p.

This means we need to take all the individual identifiers of $\mathtt{p}$ and add them to the existing identifiers of $\mathtt{c}$ (if any). In particular, $ids_c = ids_c \cup ids_p$, and $superid_c = superid_c \cup (\bigcup_{id \in ids_p} id)$. Note that only identifiers are inherited, not ordinary attributes. They remain associated only with entity objects they are locally a part of. Finally, we also need to mutually bind both the entity types with *hierarchy* morphisms. For technical reasons, in both directions. I.e., morphisms $\mathtt{c{\cdot}p{\cdot}up} = (1,1) : \mathtt{c} \to \mathtt{p}$ and $\mathtt{c{\cdot}p{\cdot}down} = (1,1) : \mathtt{p} \to \mathtt{c}$ are constructed.

Example 2.6. The translation of an ISA hierarchy in Fig. 2.3 produces entity objects `Person`, `Customer`, and `Employee`, each with respective attribute objects and morphisms. Note the inherited identifier `{PersonId}` in objects for the descendant entity types. The hierarchy morphisms correspond to the ISA relationships. □

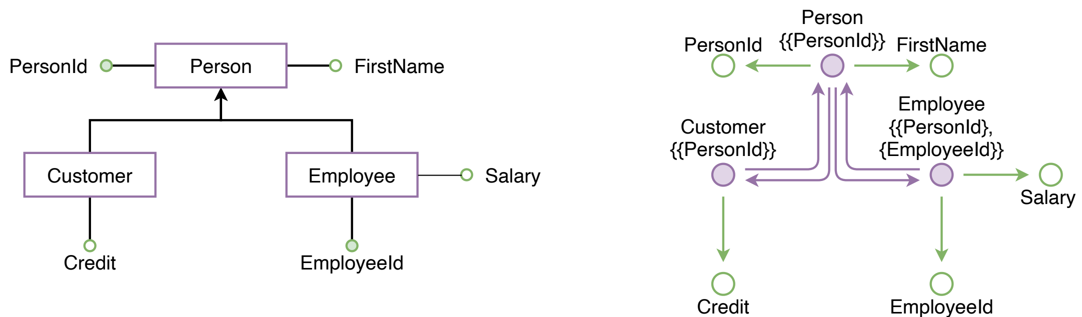


Figure 2.3: Translation of ISA hierarchies

Weak Identifiers.

Let us have an entity type e with at least one weak identifier such that all the individual entity types involved in the relationship types forming these weak identifiers are already entirely resolved (i.e., including the impact of both inheritance

and weak identifiers). In case there are more weak identifiers, let us process them one by one, each causing that the already partially resolved entity object $\mathtt{e}$ will be modified according to the following steps.

In this sense, let $\mathtt{e} = (superid, ids)$ be the current entity object for our weak entity type $e \in \mathbb{E}$ and v be a particular weak identifier we are about to transform. Let $a_1, \ldots, a_n$ for some $n \in \mathbb{N}_0$ be attributes of e (if any) involved in this weak identifier, and $r_1, \ldots, r_m$ for some $m \in \mathbb{N}$, $m \geq 1$ be all the relationship types incidental to e such that $R_v \subseteq \{1, \ldots, m\}$, $R_v \neq \emptyset$ are indices of relationship types actually involved in this weak identifier, each necessarily with cardinality $(\mathtt{1}, \mathtt{1})$ toward e. For each such relationship type r_i, $i \in R_v$ we finally assume that $e_i^1, \ldots, e_i^{k_i}$ for some $k_i \in \mathbb{N}$, $k_i \geq 2$ are all the individual entity types participating in r_i (once for each occurrence, including e itself) and for each one of them, i.e., $\forall\, j \in \{1, \ldots, k_i\}$, it then holds that (min_i^j, max_i^j) is the cardinality of e_i^j in r_i and $\mathtt{e_i^j} = (superid_i^j, ids_i^j)$ is the already fully resolved entity object for e_i^j (except for e as such).

Within the context of a particular relationship type r_i, $i \in R_v$, it may happen that some of the involved entity types may have identifiers composed of attributes with the same names (as a consequence of the inheritance of identifiers in hierarchies or in case of recursive relationship types). Similarly, such conflicts can also appear across different participating relationship types. Therefore, we will use marked attribute names instead of the original ones. For this purpose, let us introduce $m(ids_i^j) = \{\{\mathtt{a{\cdot}i{\cdot}j} \,|\, a \in id\} \,|\, id \in ids_i^j\}$ as an auxiliary set of marked identifiers for entity type e_i^j within r_i for each suitable i and j, i.e., for $i \in R_v$ and $j \in \{1, \ldots, k_i\}$ (except for e).

For each participating relationship type r_i, $i \in R_v$, let us now find all sets of attributes by which r_i may contribute to the weak identifier we are resolving. In case there exists at least one involved entity type e_i^j (except for e) with cardinality $\mathtt{1}$ in the upper boundary max_i^j, we define $ids_i = \{id \,|\, id \in m(ids_i^j) \wedge j \in \{1, \ldots, k_i\} \wedge e_i^j \neq e \wedge max_i^j = \mathtt{1}\}$. Otherwise, i.e., when $\forall\, j \in \{1, \ldots, k_i\}$ it holds that $max_i^j = *$ (except for e), we put $ids_i = \{\bigcup_{j=1}^{k_i} id^j \,|\, id^j \in m(ids_i^j) \wedge e_i^j \neq e\}$. Just for the purpose of the translation of the relationship type r_i itself, which we will cover in the next section, let us also denote all the additional (here omitted) attributes that will later on form instances of a given relationship type as $a(r_i) = \{\mathtt{a{\cdot}i{\cdot}j} \,|\, a \in id \wedge id \in m(ids_i^j) \wedge j \in \{1, \ldots, k_i\} \wedge e_i^j \neq e \wedge max_i^j = *\}$ in the former case, and $a(r_i) = \emptyset$ in the latter.

Now, we are finally ready to add all the resolved identifiers resulting from the weak identifier v between the existing ones (if any). In particular, assuming that $ids' = \{(\bigcup_{x=1}^{n} u(a_x)) \cup (\bigcup_{i \in R_v} id_i) \,|\, id_i \in ids_i\}$, we can put $ids = ids \cup ids'$ and $superid = superid \cup (\bigcup_{id \in ids'} id)$. In other words, each newly resolved identifier consists of all the local involved unfolded attributes together with marked attributes contributing from each participating relationship type.

Example 2.7. Fig. 2.4 depicts the translation of a weak entity type `Product`, resulting into an entity object `Product` with respective attribute objects and morphisms. Note how the identifier $\{\mathtt{ProductNo}, \mathtt{ManufacturerNo}\}$ was generated. □

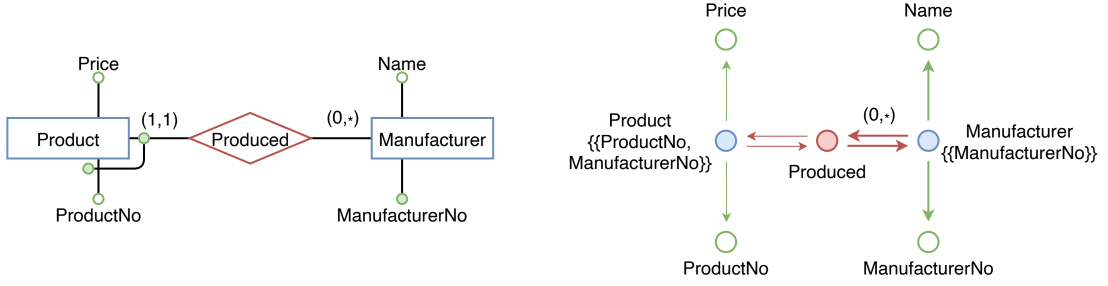


Figure 2.4: Translation of weak entity types

Relationship Types.

Let $r \in \mathbb{R}$ be a relationship type and $e_1, \ldots, e_m$ be the participating entity types (one for each individual occurrence in case of recursive relationship types) for some $m \in \mathbb{N}$, $m \geq 2$. Let each of these entity types $e_i \in \mathbb{E}$ be associated with a cardinality (min_i, max_i), and $\mathtt{e_i} = (superid_i, ids_i)$ be the corresponding fully resolved entity object for e_i. Finally, let $a_1, \ldots, a_n$ be attributes associated with r (if any) for some $n \in \mathbb{N}_0$, each $a_i \in \mathbb{A}$. For analogous reasons as in the previous section, let $n(ids_i) = \{\{\mathtt{a} \cdot \mathtt{i} \,|\, a \in id\} \,|\, id \in ids_i\}$ for any i. *Relationship* object $\mathtt{r} = (superid, ids)$ for r is constructed as follows.

If r participates in a weak identifier for some e_i (necessarily only at most one such e_i), we put $ids = ids_i$ and $superid = superid_i \cup a(r)$, as defined in the previous section.

Else, if there exists at least one entity type e_i such that $max_i = \mathtt{1}$, we define $ids = \{id \,|\, id \in n(ids_i) \wedge i \in \{1, \ldots, m\} \wedge max_i = \mathtt{1}\}$. Otherwise, i.e., when $\forall\, i \in \{1, \ldots, m\}$ it holds that $max_i = *$, we put $ids = \{\bigcup_{i=1}^{m} id_i \,|\, id_i \in n(ids_i)\}$, i.e., identifiers of all the involved relationship types need to be incorporated. Under all circumstances, $superid = \bigcup_{i=1}^{m} \bigcup_{id_i \in n(ids_i)} id_i$ so that instances of $\mathtt{r}$ in the instance category can be correctly represented and fully materialised.

Next, we need to interlink the created relationship object $\mathtt{r}$ with every entity object $\mathtt{e_i}$. I.e., for $\forall\, i \in \{1, \ldots, m\}$, two relationship morphisms are created, one for each direction. In particular, $\mathtt{r} \cdot \mathtt{e_i} \cdot \mathtt{in} = (min_i, max_i) : \mathtt{e_i} \rightarrow \mathtt{r}$, and $\mathtt{r} \cdot \mathtt{e_i} \cdot \mathtt{out} = (\mathtt{1}, \mathtt{1}) : \mathtt{r} \rightarrow \mathtt{e_i}$. Finally, attributes $a_1, \ldots, a_n$ (if any) associated with the relationship type $\mathtt{r}$ as such will be transformed exactly the same way as in case of attributes associated with entity types.

Example 2.8. Relationship type `Order` between two entity types `Person` and entity type `Product` depicted in Fig. 2.5 is transformed to a relationship object `Order` and a set of relationship morphisms, together with respective cardinalities, connecting the relationship object `Order` with the respective entity objects `Person` and `Product`. □

Overall Example

In Fig. 2.6, we depict the whole translated schema category for the ER schema of the multi-model scenario from Fig. 2.1.

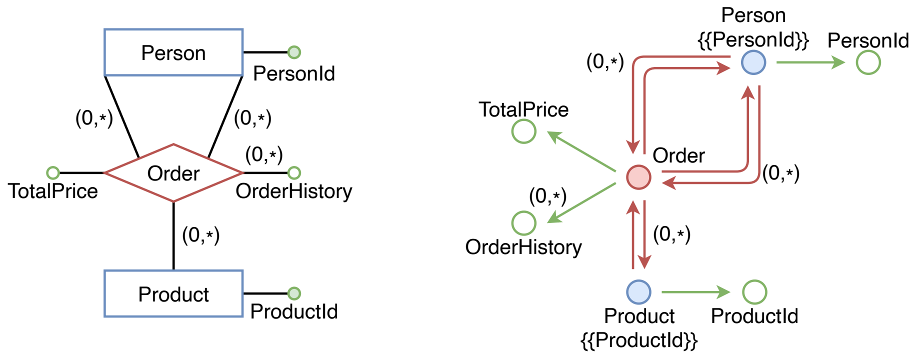


Figure 2.5: Translation of relationship types

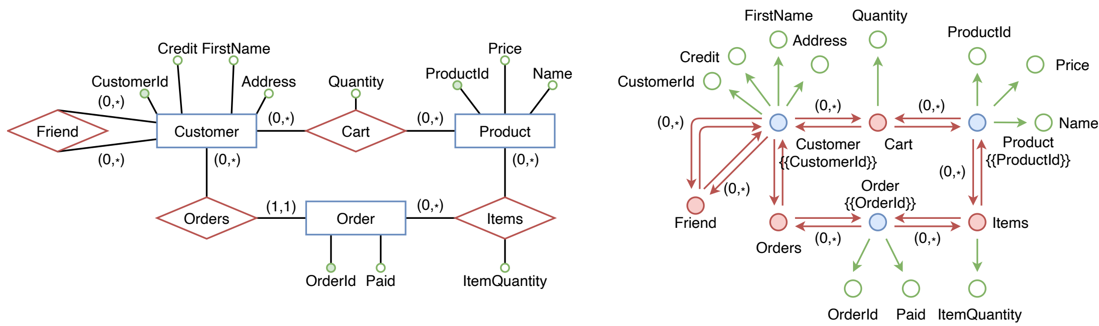


Figure 2.6: Translation of the whole ER schema

2.5 Categorical Framework

Having described the key building blocks, we can introduce the concept of the whole multi-model categorical framework. Since the idea is complex and we are at the beginning of this research task, we provide an overall picture of the architecture and a general description of the idea. Formal definitions, algorithms, and implementation form our near-future work.

As depicted in Fig. 2.7, the whole framework involves three related parts. On the left, there is the input ER schema of the reality as understood by the user. It can then be automatically translated (see Section 2.4.3) to the schema category (see Section 2.4.1), and, having the particular data instances, also the instance category (see Section 2.4.2). The schema category is the core part of the framework, being an abstraction for the unified representation of the (combination of) logical model(s). The instance category is an auxiliary data structure, not just for the purpose of, e.g., representation of (intermediate) results of queries.

As also depicted in the figure by the colours, the user is expected to denote (either in the original ER schema or in the schema category graph) the (possibly overlapping) parts corresponding to particular data models. The category is then mapped to a particular logical multi-model schema. Implementation details of such a transformation strongly depend on the selection of the particular multi-model DBMSs, and, thus, are beyond the scope of this paper. Intuitively, e.g., for the relational model, we can create a table for each entity and a relationship object consisting of the related single-value attribute objects, separate tables for multi-valued attributes, and then merge the tables where the cardinalities allow. Or, the document model can be transformed using a BFS traversal starting with

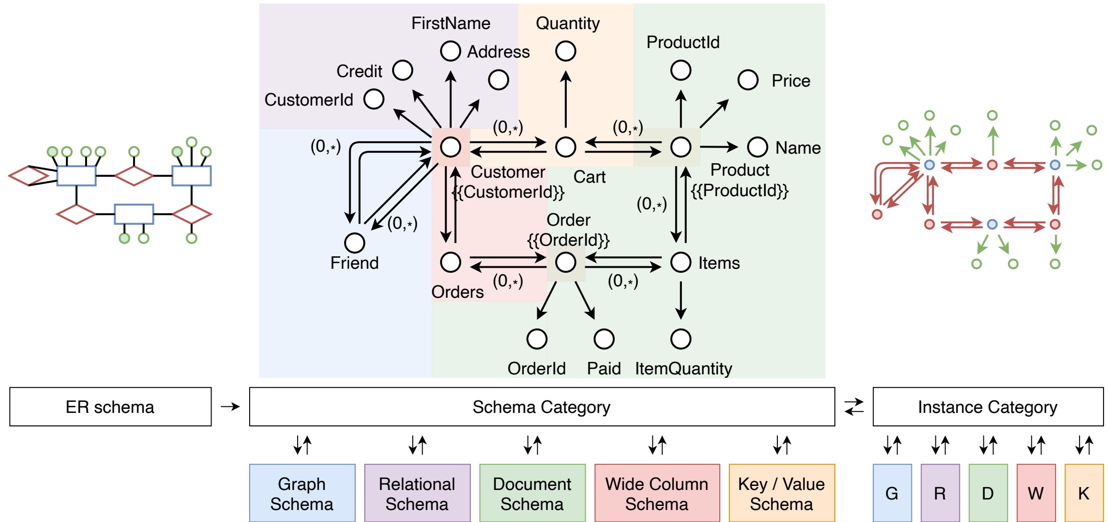


Figure 2.7: Architecture of the categorical framework

a selected object. An inspiration can be found, e.g., for the relational model in [134] or for object-relational model in [19].

2.6 Proposal Evaluation

The main added value of the described categorical approach is that we enable the user to connect the conceptual schema with the logical layer of a particular multi-model DBMSs. As a consequence, we also gain the following main outcomes:

- Since the categorical conceptual layer is defined universally for any data model, it can then be applied to any multi-model DBMSs. In addition, the idea enables us to cover even data models that are not currently known; the only requirement is that they can be described using the same idea.

- As described before, the user specifies which part of the categorical schema is mapped to a particular logical model. However, such a specification of models does not have to be disjunctive. They can overlap arbitrarily, hence the data may be stored redundantly, but the principles remain unchanged.

- At the categorical conceptual layer, we can define not only data structures, but also a conceptual query language accessing/manipulating the data independently of a chosen DBMS (its query language, specific syntax, etc.). The language can work with the categorical graph, e.g., exploiting well-known means of graph query languages. Using a similar strategy, the conceptual queries can be translated to expressions required by a particular DBMS.

- The transformation of one model (or its part) to another model can be defined precisely using the category theory. Therefore, we have a mathematical background for various aspects of evolution management, such as data modification, data migration between the models, or even self-tuning of the DBMS reflecting changes in the user interaction.

The proposed approach first requires a non-trivial effort in both theoretical and development directions. The remaining open questions and challenges that need to be resolved in order to create a full-fledged solution comparable, e.g., to the approaches used for the traditional relational DBMSs, are as follows:

- *Conceptual modelling*: The described categorical model might need to be further extended, e.g., with data types for attribute values. A more complex point to discuss is whether entity types, attributes, and relationship types actually need to be distinguished at all.
- *Conceptual querying*: The conceptual query language needs to be defined so that it can be mapped to any multi-model query language. On the other hand, it should not be too different from the existing popular query languages. Since the categorical model is backed by a graph, the query language might be inspired by graph query languages like Cypher or SPARQL.
- *Inter-model transformations*: The inter-model transformations need to be defined in order to cover all the possible cases and support correct and complete evolution management. Similarly to the case of querying, a language that would enable to express the required changes needs to be defined, too.
- *Compact representation*: In the proposed approach, we assumed that the user starts to design the application using an existing verified tool, such as ER or UML. However, the categorical model can also be designed directly. For this purpose, we would, however, need a compact version that enables more user-friendly expressions and structures.
- *Reverse engineering*: An important related issue to be dealt with is reverse engineering. We can encounter a situation when new data needs to be integrated into an existing multi-model scenario with the whole categorical framework already built. Hence, we need to infer the structure (schema) of the input data and map it to the existing categorical schema, respectively.
- *Robustness*: In general, the core idea might need to be further extended in order to provide a sufficient level of robustness to ensure the above-described benefits (redundancy, data migration, query update, etc.) to a full extent.

2.7 Conclusion

Category theory brings a strong formalism that enables to model various data structures. In this paper, we introduced its exploitation for the purpose of unified conceptual modelling and representation, and, as a consequence, correct and efficient management of multi-model data. As the task is, in general, quite complex, we do not provide a fully functional solution but rather a research direction, which we hope will further be extended and tackled by the database community.

Paper III

A Unified Representation and Transformation of Multi-Model Data using Category Theory

Pavel Koupil[@1], Irena Holubová[1]

Published in Journal of Big Data (9, 61, 2022) by Springer Nature
doi: 10.1186/s40537-022-00613-3

[@] corresponding author, e-mail: pavel.koupil@matfyz.cuni.cz

[1] 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

3.1 Introduction

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

Example 3.1. An example of a multi-model scenario is provided in Figure 3.1. 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." [140] □

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* [8], 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* [13] that offer the support of multiple models under the hood of a single system, treating all the data models as first-class citizens [12]. 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 [145] 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 3.2. In Figure 3.2, we depict an ER schema of the multi-model scenario from Figure 3.1. □

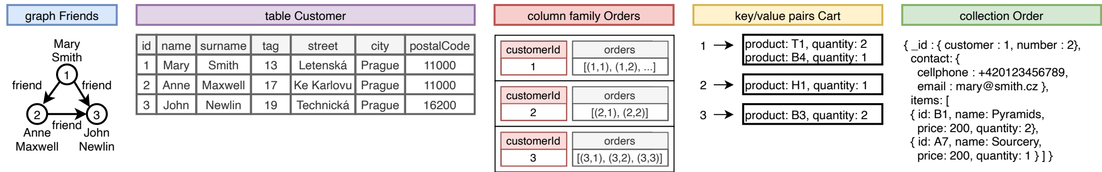


Figure 3.1: A sample multi-model scenario

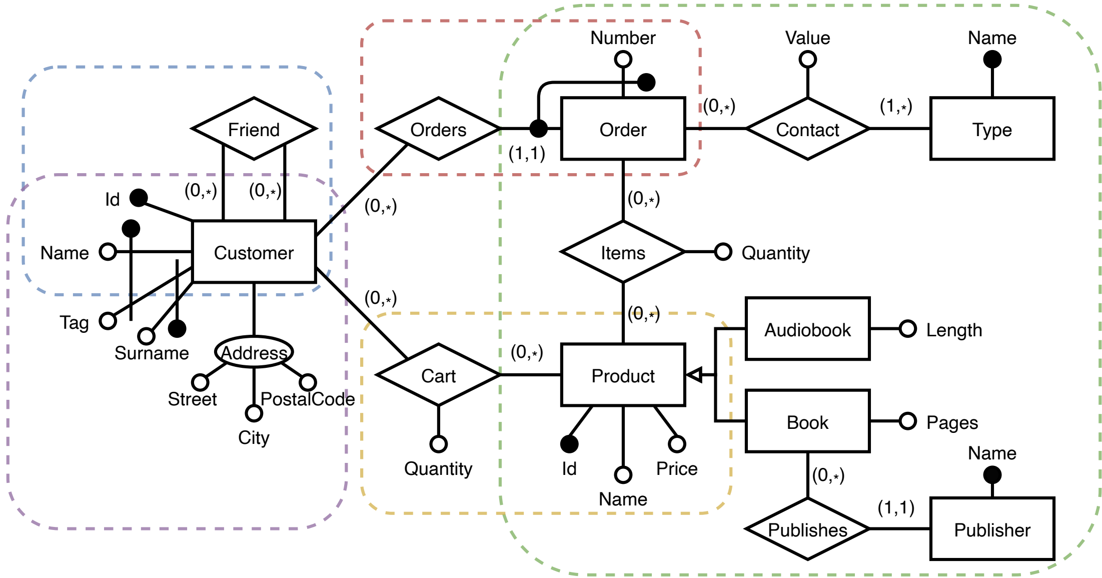


Figure 3.2: ER schema of the sample multi-model scenario in Figure 3.1

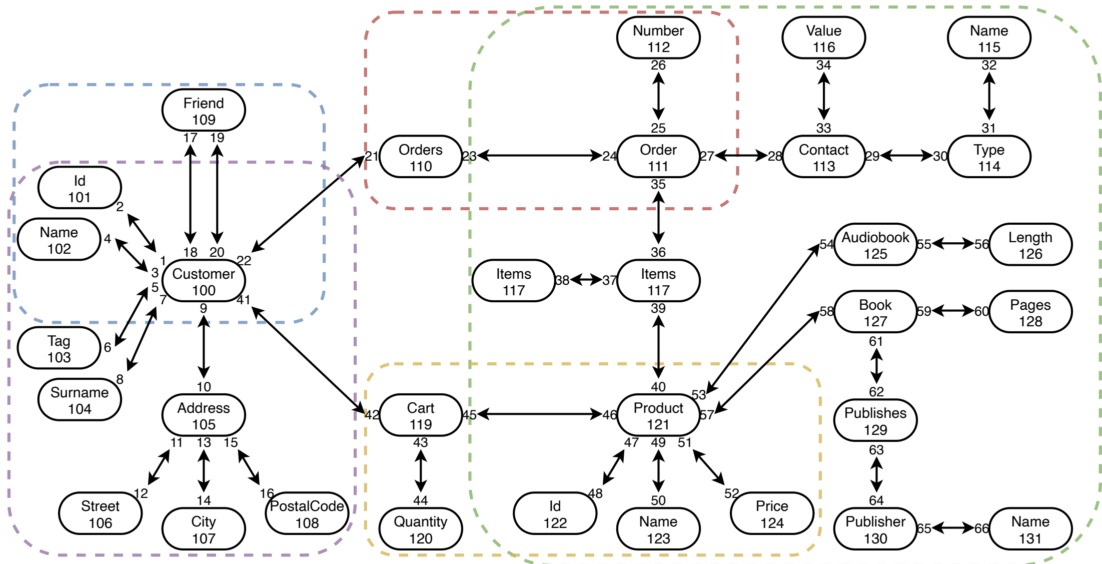


Figure 3.3: Schema category extracted from the sample ER schema in Figure 3.2

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 standardised) 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 reorganisations; 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 [147], 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 [2], we have proposed a solution based on *category theory* [133], "*the most general and abstract branch of pure mathematics*" [148] 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 Section 3.2, 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. Section 3.3 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 Section 3.4, 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 Section 3.5, 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 Section 3.6 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 Section 3.7 we overview the related work and its drawbacks reflected in our approach. We conclude and outline future work in Section 3.8.

3.2 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 semi-structured 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 3.1. (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. Hence, 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,

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

Table 3.1: 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
Array	–	–	Array	–	Array	Array	Array
Structure	–	–	–	–	Set / ZSet / Hash	Nested document	Super 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	–

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., represent 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-normalisation.

- *Aggregate-ignorant models* (graph, relational, RDF, array): These models are not primarily oriented to the support of aggregates. The relational world strongly emphasises the normalisation 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.

[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.

3.2.1 Basic Concepts of Category Theory

Category theory is a branch of mathematics that attempts to formalise 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 visualised in the form of a multigraph, where category objects act as vertices and category morphisms as directed edges.

Each morphism is modelled 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 3.3. In Figure 3.4, 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. □

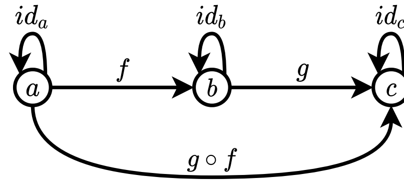


Figure 3.4: An example of a category

Example 3.4. **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

[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.

context of our approach, it is worth focusing on categories derived from graphs, as well as categories built on top of other categories.

Example 3.5. 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.

3.2.2 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 [2]. 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 modelling 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 modelled 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 modelled 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, $min \in \{\mathtt{0}, \mathtt{1}\}$ and $max \in \{\mathtt{1}, *\}$ 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 $1_o = (\varepsilon,\ o,\ o,\ \mathtt{1},\ \mathtt{1})$. 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 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 $m : X \rightarrow Y$ is a non-identity morphism between two particular objects, there must also exist its *dual* morphism $m^{-1} : Y \rightarrow X$. 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 [2] 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

[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.

objects, depending on the respective data model it represents. If a schema category **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 3.6. The schema category of ER model in Figure 3.2 is depicted in Figure 3.3. Each object is represented as a node labelled with *key* and *label*. Morphisms are represented as directed edges labelled 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]

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:

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

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

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

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

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

Relationship object *Publishes* has the following categorical representation:

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

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 **S** is to describe the structure of the data at the conceptual layer, *instance category* **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 **S**. It permits us to encompass all data valid against **S** stored in the database at a selected moment. When the data is modified (within the restrictions given by **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 **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 modelled as a set of

[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.

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 3.7. An instance object $Customer_\mathbf{I}$ for $Customer_\mathbf{S}$ from Figure 3.3 can, for example, be:
$Customer_\mathbf{I} = \{$
$\{(\mathtt{1}, 1), (\mathtt{3}, Mary), (\mathtt{5}, 13), (\mathtt{7}, Smith)\},$
$\{(\mathtt{1}, 2), (\mathtt{3}, Anne), (\mathtt{5}, 17), (\mathtt{7}, Maxwell)\},$
$\{(\mathtt{1}, 3), (\mathtt{3}, John), (\mathtt{5}, 19), (\mathtt{7}, Newlin)\}\}$
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 3.4). 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 $|\{t_2 \,|\, t_2 \in o_2, (t_1, t_2) \in m_\mathbf{I})\}| = c$ 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) $1_o = \{(t, t) \,|\, t \in o\}$. The composition operation $\circ_\mathbf{I}$ corresponds to the composition in **Rel**.

Example 3.8. Consider object $Customer_\mathbf{I}$ from Example 3.7 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]
$7_\mathbf{I} = \{$
$(\{(\mathtt{1}, 1), (\mathtt{3}, Mary), (\mathtt{5}, 13), (\mathtt{7}, Smith)\}, \{(\epsilon, Smith)\}),$
$(\{(\mathtt{1}, 2), (\mathtt{3}, Anne), (\mathtt{5}, 17), (\mathtt{7}, Maxwell)\}, \{(\epsilon, Maxwell)\}),$
$(\{(\mathtt{1}, 3), (\mathtt{3}, John), (\mathtt{5}, 19), (\mathtt{7}, Newlin)\}, \{(\epsilon, Newlin)\})\}$ □

Having a schema category **S** and a particular instance category **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.

3.3 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

[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.

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 visualise and process schema categories, e.g., using a tool called *MM-cat* which we introduce in Section 3.5.

This section aims to describe how the mappings are intended to be created and formalised. After an informal outline of the basic principles, we provide a formal definition of these mappings (Section 3.3.1), and we introduce an alternative way that these mappings can be visualised or even directly created by users using a textual representation (Section 3.3.2).

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 neighbouring or even more remote objects or morphisms – are stored as individual records within a given kind (recall Table 3.1) 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 neighbour 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 3.9. Consider the document collection *Order* in Figure 3.1. For example, properties *customer* or *price* have user-defined names (different from the ones used in the ER schema and schema category in Figures 3.2 and 3.3). 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 3.10. A sample decomposition is presented in Figure 3.2 and Figure 3.3 using the colours from Figure 3.1. □

3.3.1 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.

[10]If required by the respective model.

- 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.

Example 3.11. 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:

- A *simple value* is a signature of morphism $m_{value} : o_\phi \to o_{value}$, where o_{value} is the object representing the simple values.
- An *array* is 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.

3.3.2 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 Figure 3.5. `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

```
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)*
```

Figure 3.5: Grammar of the JSON-like representation of access paths

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 3.12. Figure 3.6 illustrates the access path of collection *Order* from Figure 3.1. 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*.

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. □

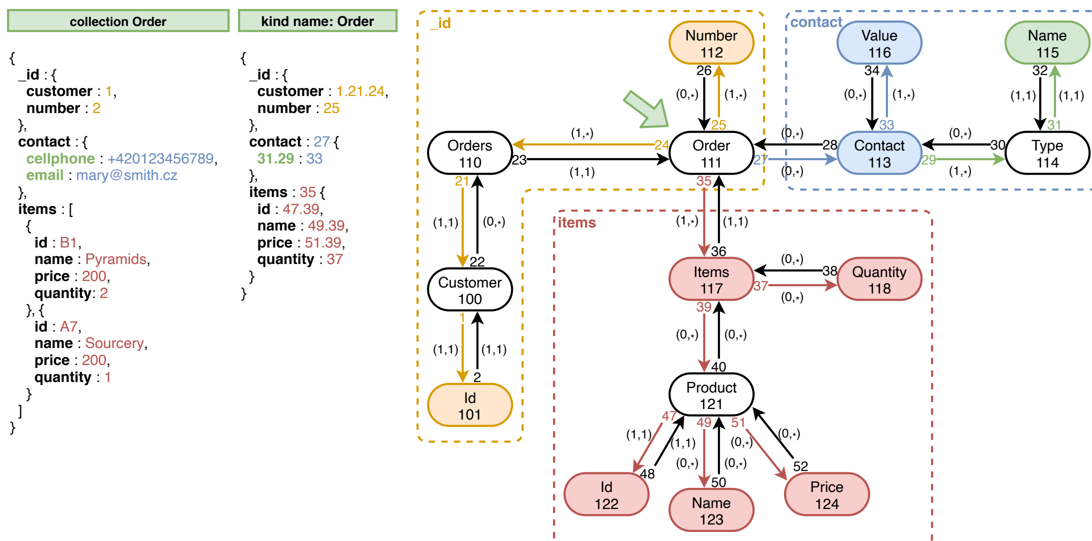


Figure 3.6: Collection *Order*, an access path for kind *Order*, and the corresponding part of schema category **S**

Note that there is a difference between aggregate-ignorant and aggregate-oriented models. For aggregate-ignorant ones, there is no need to consider ARRAY

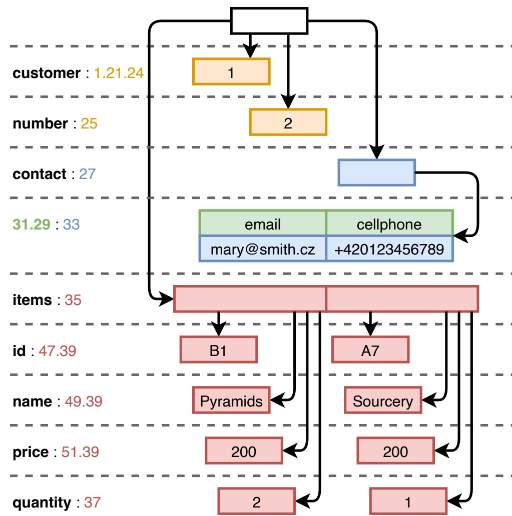


Figure 3.7: Forest containing a single record corresponding to a document

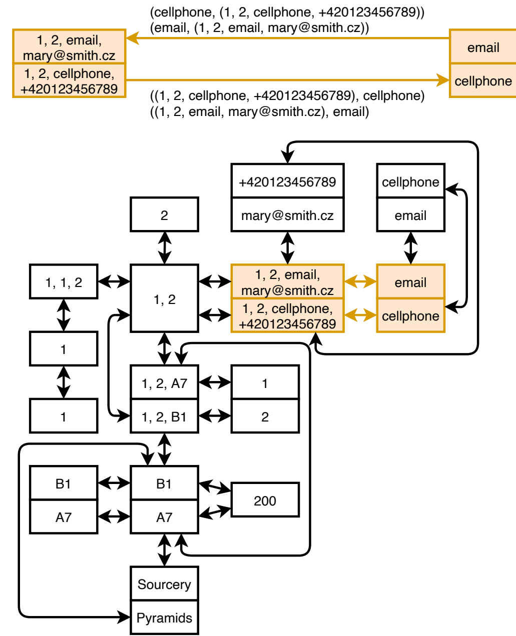


Figure 3.8: Part of instance category **I** after the insertion of single Order

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.

3.4 Transformations

Having defined the schema category **S**, instance category **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.

3.4.1 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 **I**.

Forest of Records

To be able to uniformly manipulate records from different data models (recall Table 3.1), 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 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 materialise the whole forest for all input trees. Only the currently processed data fragments are constructed for further processing.)

Example 3.13. Figure 3.7 illustrates the representation of document *Order* corresponding to the access path depicted in Figure 3.6 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

[11]Depending on the particular model.

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 3.14. As illustrated in Figure 3.9, 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). □

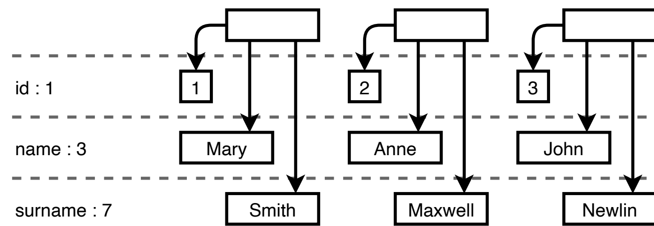


Figure 3.9: Forest containing three records corresponding to a row of a relational table

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 Section 3.5 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 3.1.

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 $Inst_\mathbf{I} : \mathbf{S} \rightarrow \mathbf{I}$. 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 3.15. Consider again Figure 3.7 and 3.6. 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 $\mathbf{S}$ 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 115.) 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 $m_\mathbf{I} : p_\mathbf{I} \to o_\mathbf{I}$ 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.,

Algorithm 3.1: Model-to-Category Transformation

Input: $\mathbf{S}$ – schema category
1. $\mathbf{I}$ – instance category
2. $F_\kappa = (T_\kappa, M_\kappa)$ – forest of input records
3. P_κ – access path associated with κ
4. $root_\kappa$ – root object associated with κ
5. $morph_\kappa$ – root morphism associated with κ
6. $M \leftarrow$ an empty stack
7. **foreach** record r in T_κ **do**
    - `// preparation phase:`
8. **if** $morph_\kappa$ is $null$ **then**
        - `// κ with root object:`
9. $q_\mathbf{I} := Inst_\mathbf{I}(root_\kappa)$
10. $S :=$ fetchSids($root_\kappa.superid$, r, null)
11. **foreach** sid in S **do**
12. $sid :=$ modifyActiveDomain($q_\mathbf{I}$, sid)
13. **foreach** ($context$, $value$) in children(P_κ) **do**
14. M.push((sid, $context$, $value$))
15. **else**
        - `// κ with root morphism:`
16. $S_{dom} :=$ fetchSids($root_\kappa.superid$, r, null)
17. $sid_{dom} :=$ modifyActiveDomain($Inst_\mathbf{I}(root_\kappa)$, $S_{dom}.get(0)$)
18. $q_{cod} := morph_\kappa.cod$
19. $S_{cod} :=$ fetchSids($q_{cod}.superid$, r, null)
20. $sid_{cod} :=$ modifyActiveDomain($Inst_\mathbf{I}(q_{cod})$, $S_{cod}.get(0)$)
21. $m_\mathbf{I} := Inst_\mathbf{I}(morph_\kappa)$
22. addRelation($m_\mathbf{I}$, sid_{dom}, sid_{cod}, r)
23. addRelation($m_\mathbf{I}^{-1}$, sid_{cod}, sid_{dom}, r)
24. $t_{dom} :=$ getSubpathBySignature(P_κ, $null$)
25. $t_{cod} :=$ getSubpathBySignature(P_κ, $morph_\kappa$)
26. **foreach** ($context$, $value$) in (children(P_κ) $\setminus \{(t_{dom}, t_{cod})\}$) **do**
27. M.push((sid_{dom}, $context$, $value$))
28. **foreach** ($context$, $value$) in children(t_{cod}) **do**
29. M.push((sid_{cod}, $context$, $value$))
    - `// processing of the tree:`
30. **while** M is not empty **do**
31. (pid, $m_\mathbf{S}$, t) $:= M$.pop()
32. $m_\mathbf{I} := Inst_\mathbf{I}(m_\mathbf{S})$
33. $o_\mathbf{S} := m_\mathbf{S}.cod$
34. $q_\mathbf{I} := Inst_\mathbf{I}(o_\mathbf{S})$
35. $S :=$ fetchSids($o_\mathbf{S}.superid$, r, pid)
36. **foreach** sid in S **do**
37. $sid :=$ modifyActiveDomain($q_\mathbf{I}$, sid)
38. addRelation($m_\mathbf{I}$,pid, sid, r)
39. addRelation($m_\mathbf{I}^{-1}$,sid, pid, r)
40. **foreach** ($context$, $value$) in children(t) **do**
41. M.push((sid, $context$, $value$))

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 $\mathbf{I}$, i.e., internal structures of objects and morphisms in $\mathbf{I}$ are appropriately extended.

Example 3.16. Suppose that we have an access path depicted in Figure 3.6 and a corresponding forest of records depicted in Figure 3.7. The intended transformation should convert the data represented in the document model to the categorical representation corresponding to the schema category $\mathbf{S}$ depicted in Figure 3.3 and non-empty instance category $\mathbf{I}$.

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 $\mathbf{S}$. Next, instance category $\mathbf{I}$ 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_{κ} (depicted in Figure 3.6) is traversed, creating triples for stack M for property *Customer*, *Number*, *Items*, and *Contact* related to *sid*, as depicted in Figure 3.10. In the figure on the right we can also see the current content of instance category $\mathbf{I}$, 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 }
```

associated with pid ($customer : 2, number : 1$) and morphism $35_{\mathbf{I}} : Order_{\mathbf{I}} \rightarrow Items_{\mathbf{I}}$ of instance category $\mathbf{I}$. 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 Figure 3.11. The algorithm continues in the same way until stack M is empty. The resulting part of the instance category $\mathbf{I}$ corresponding to kind *Order* is depicted in Figure 3.8. □

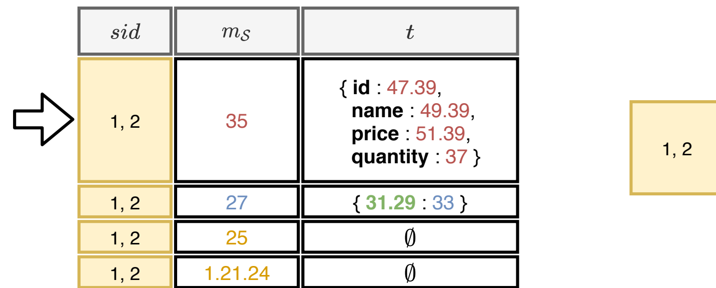


Figure 3.10: Stack M and instance category $\mathbf{I}$ when the preparation phase for data from Figure 3.7 is completed

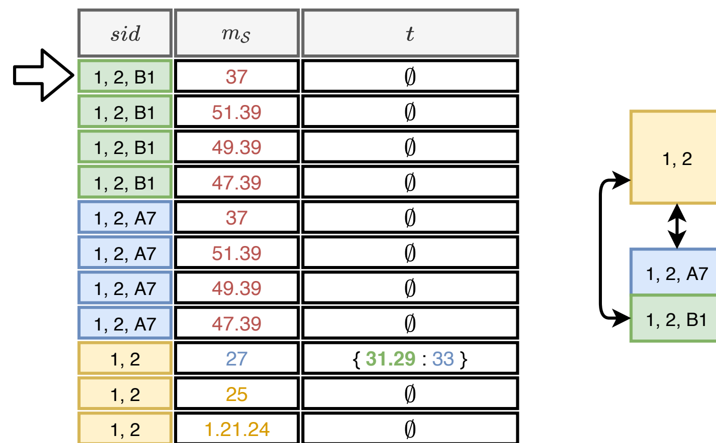


Figure 3.11: Stack M and instance category $\mathbf{I}$ when the first iteration of tree processing completed

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 3.2) 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 3.2: 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
```

Algorithm 3.3: 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)
```

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 3.3. 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.

3.4.2 Category-to-Model Transformation

Having an instance category **I** and mapping $\mathfrak{M}$, the opposite direction of transformation allows extraction of data from **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 Section DDL Algorithm).
2. *DML Algorithm*: Transformation of data instances from instance category **I** to a particular logical model (see Section DML Algorithm).
3. *IC Algorithm*: Finalisation of schema definition with integrity constraints, i.e., adding of identifiers and references to other kinds (see Section IC Algorithm).

DDL Algorithm

Having a schema category **S**, instance category **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 visualised 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 subpath 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 3.4, we begin the processing with the setting of kind name $name_\kappa$ to wrapper W_D and we check whether the schema is applicable,

Algorithm 3.4: 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()
```

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.[12] Otherwise, traversing of the access path P_κ is carried out using stack M. It is initialised 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 *deter-*

[12]For example, in *MongoDB* this would be command `db.createCollection("orders")`.

minePropertyName() 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.[13] 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.

DML Algorithm

Having the schema category $\mathbf{S}$, instance category $\mathbf{I}$, 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 3.4 with data from instance category $\mathbf{I}$.

As we can see in Algorithm 3.5, we first initialise 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

[13] 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.

Algorithm 3.5: 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
```

we get access subpath t_{cod} of codomain $morph_\kappa.cod$ using function *getSubpath-BySignature()*. For each $s \in S$ we initialise 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 3.6, function *buildStatement()* iteratively processes the initialised 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 $(name, value)$[14] of data from

[14]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.

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.

Algorithm 3.6: 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 3.7 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 3.7: 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()

3.4.3 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 Figure 3.12, we can see a part of the schema category of the sample data. The colours 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 Section 3.4.1) whose result is provided in Figure 3.8, 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 Section 3.4.2). First, it creates the respective schemas (see Section DDL Algorithm). 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 Section DML Algorithm) and in the last step it adds the respective integrity constraints (see Section IC Algorithm), 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.

3.5 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* [64]. Its primary purpose is user-friendly modelling 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

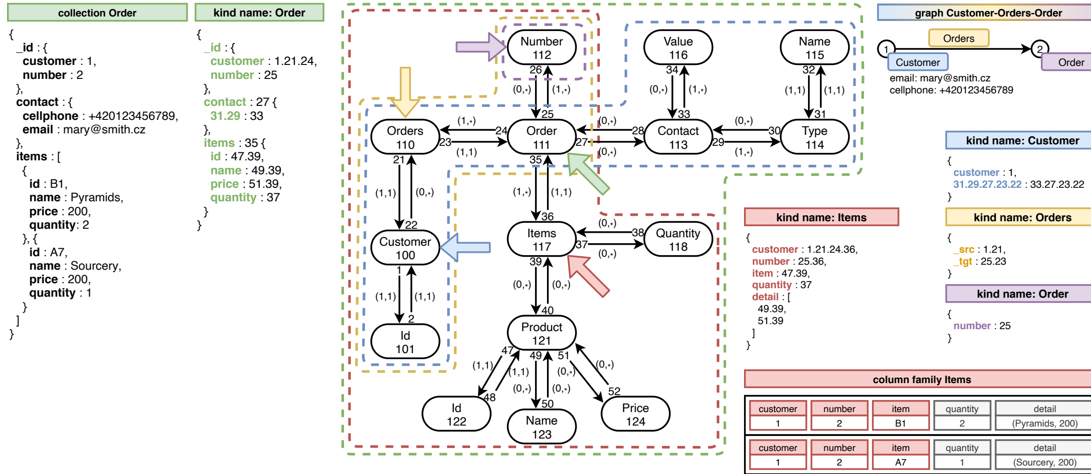


Figure 3.12: Example of transformation from document model to graph and column models

possible extensions and application of the core idea in advanced data processing tasks forming our current and near-future work as discussed in Section 3.8.

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 $\mathbf{S}$ using the algorithm proposed in [2].[15]

3. $\mathbf{S}$ 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[16] 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 $\mathbf{S}$ 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 visualised 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, the database structures (i.e., tables, collections etc.) in particular DBMS(s) as well as instance category $\mathbf{I}$ in *MM-cat* are empty.

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

[15] An advanced user can create $\mathbf{S}$ directly, without the need to create the ER schema. We add this auxiliary step for easier understanding through a well-known approach.

[16] The combined models can be a part of separate DBMSs or a single multi-model DBMS.

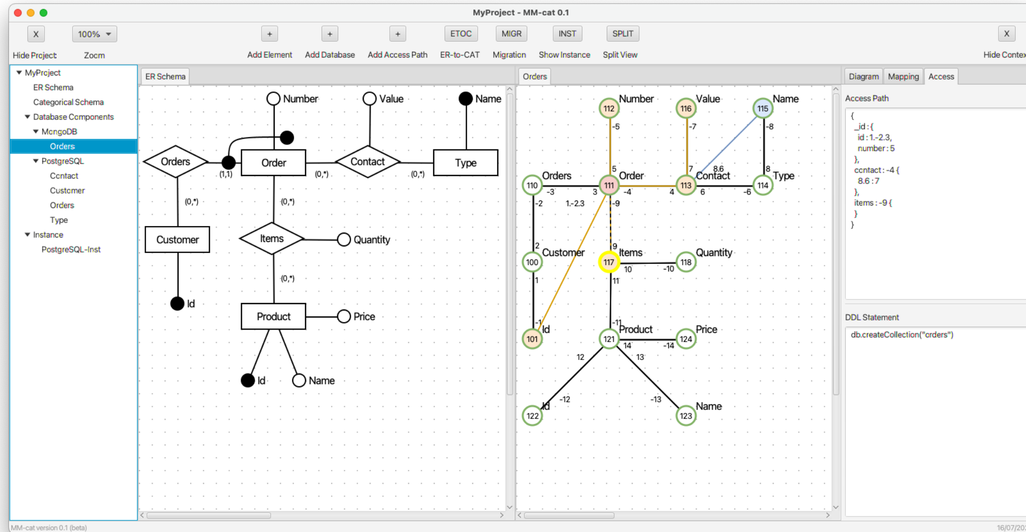


Figure 3.13: Sample screen shot of *MM-cat*

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 Figure 3.13, *MM-cat* enables us to visualise and modify the current status of the multi-model modelling 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 modelling – *MongoDB*[17] and *PostgreSQL*[18]. The versatility of the approach can be demonstrated from different viewpoints:

1. *Schema-less (MongoDB)*[19] *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 modelled 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

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

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

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

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 modelled schema*: During the modelling 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.

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 modelling 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.

3.5.1 Architecture and Implementation

MM-cat was implemented using *Java SE 16*, graphical library *JavaFX*,[20] and *Apache Maven*.[21] For communication with *MongoDB* and *PostgreSQL* we use the respective Java (JDBC) drivers.[22][23]

The architecture of the framework is depicted in Figure 3.14. 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.

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

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

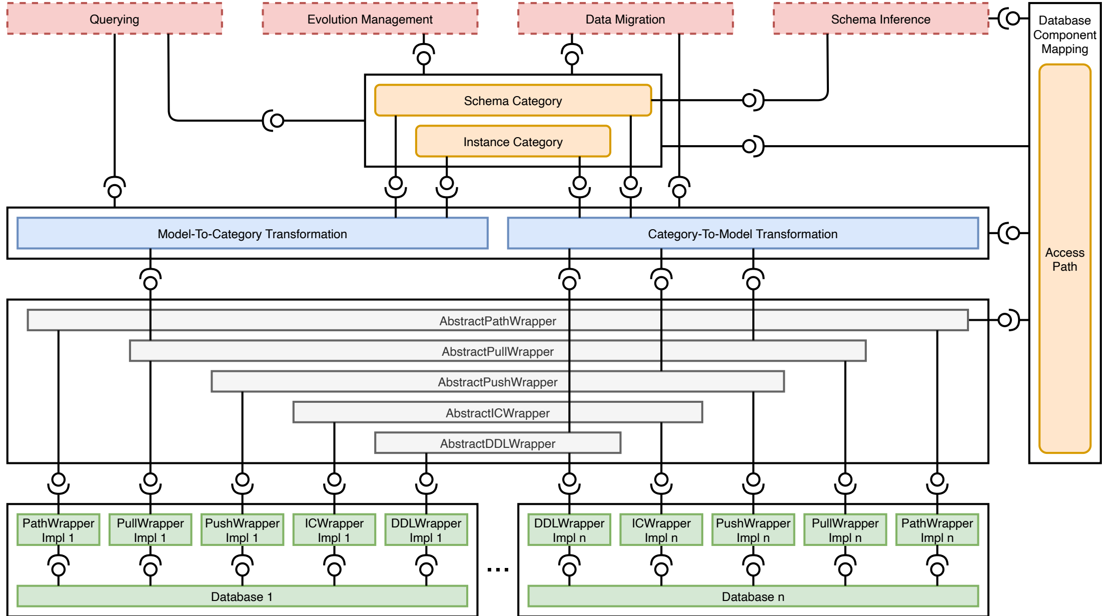


Figure 3.14: Architecture of *MM-cat*

(green) wrapper implements an interface of the respective abstract (grey) wrapper. The yellow boxes represent the core categorical data structures defined in Sections 3.2.2 and 3.3, 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 Section 3.4) 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.

`AbstractPathWrapper` From the point of view of the proposed categorical representation, the most interesting wrapper is `AbstractPathWrapper`. As we can see in Table 3.2, it returns information about the allowed complexity of the mapping in the particular DBMS.

Table 3.2: 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
`isGroupingAllowed()`	True	False
`isDynamicNamingAllowed()`	True	False
`isAnonymousNamingAllowed()`	True	False
`isReferenceAllowed()`	True	True

For example, in the case of *MongoDB* its `MongoDBPathWrapper`[24] 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 $\mathbf{S}$.

[24]Which implements `AbstractPathWrapper`

For *PostgreSQL*, its `PostgreSQLPathWrapper`[24] 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 3.4 (DDL Algorithm) are predefined by the `AbstractDDLWrapper`. In particular they involve the following ones:

- 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 optional) throws UnsupportedOperationException` creates an array of simple data types.[25]
- Method `addComplexProperty(Set<String> names, boolean optional) throws UnsupportedOperationException` creates a property with a complex type (structure).
- Method `addComplexArrayProperty(Set<String> names, boolean optional) 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.)

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

- Method `createDDLStatement()` creates and returns the resulting DDL command for a particular DBMS.

For instance, `MongoDBDDLWrapper`[26] implements only method `setKindName()`, whereas the remaining ones are empty (because *MongoDB* is schema-less) and do not throw any exception (because *MongoDB* is aggregate-oriented). Method `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 3.5 (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 underlying database systems implement methods `setKindName()`, `append()`, and `clear()` in the same way. Naturally the key difference is in method `createDMLStatement()` which is strongly system-dependent. For example, *PostgreSQL* wrapper transforms pairs ($\texttt{name}_1$, $\texttt{value}_1$), ..., ($\texttt{name}_n$, $\texttt{value}_n$) of kind κ to SQL command `INSERT INTO KIND` κ ($\texttt{name}_1$, ..., $\texttt{name}_n$) `VALUES` ($\texttt{value}_1$, ..., $\texttt{value}_n$). On the other hand, *MongoDB* wrapper creates command `db.collections.insert(...)`, where $\texttt{name}_i$, $\texttt{i} = 1, ...n$ denote names of fields in the hierarchy and $\texttt{value}_i$ denote their respective values.

`AbstractICWrapper` Methods used in Algorithm 3.7 (IC Algorithm) are predefined by the `AbstractICWrapper`. In particular they involve the following ones:

- Method `appendIdentifier(String name, IdentifierStructure idst)` enables the addition of an integrity constraint representing an identifier to kind specified using parameter `name`. The structure of the identifier

[26] Which implements `AbstractDDLWrapper`

is provided in parameter `idst`. 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 nameRef, String nameTo, Set<Pair <String, String>> atts)` enables adding of a reference from referencing kind specified using parameter `nameRef` to referenced kind specified using parameter `nameTo`. 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 schema-less *MongoDB* the references do not modify the schema at all.

`AbstractPullWrapper` Last but not least, `AbstractPullWrapper` predefines the methods used in Algorithm 3.1 (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 3.1) can be parallelised 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*[27] 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 parallelised DFS algorithm as every subtree of the access path is independent of its sibling subtrees.

- DDL algorithm 3.4 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 3.5 is parallelisable depending on whether $morph_\kappa$ is *null* or not. If it is *null*, then line 13 can be parallelised, 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 3.7 only generates statements of type `ALTER KIND`, therefore it is not considered being parallelised.

The still gradually improved implementation of the approach, *MM-cat*, contains various technical tricks enabling further optimisation. 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 synchronisation, 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 synchronise 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 materialised into a mapping. We use a lazy strategy, where only a repeatedly used composite morphism is materialised.

3.6 Benefits of Category Theory

To conclude the description of the proposed approach, we discuss the main benefits of the utilisation 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

[27] https://docs.oracle.com/en/java/javase/17/docs/api/

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)normalisation of data. More abstract data representations also exist, however, their expressive power is still limited as we discuss in [26].

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 [1], 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.

3.6.1 Application - Querying

To demonstrate how the categorical querying over the proposed categorical framework could work, let us consider the following sample multi-model query [140]: "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, Figure 3.15 depicts a possible schema of the projected properties including its mapping to the output graph model representation (hence depicted in the blue colour). 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 labelled 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.

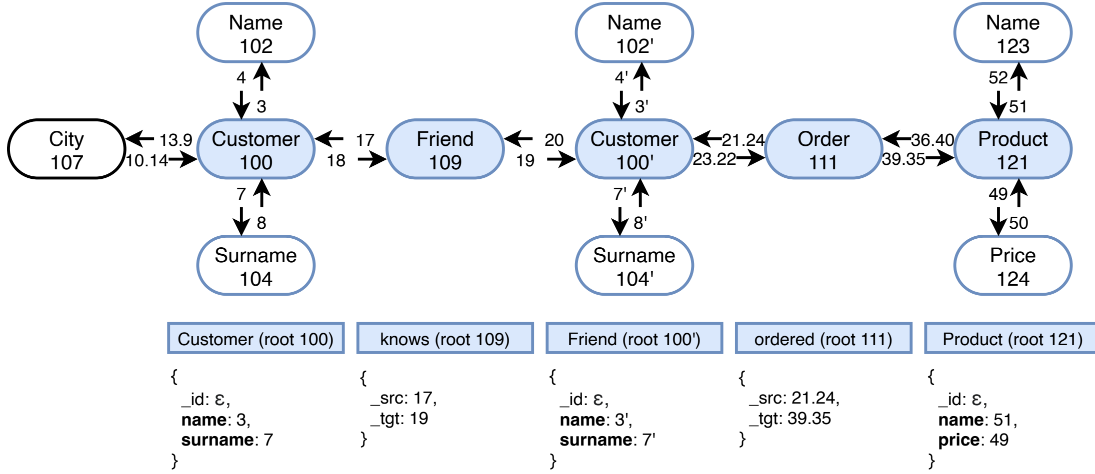


Figure 3.15: Projection of the result corresponding to the graph pattern Customer – knows → Friend – ordered → Product and its mapping to the graph model

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 [1], 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 Figure 3.15, 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 [54]. In comparison to, e.g., *Cypher* [149] 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., $39.35.23.22 : Customer \rightarrow Product$ can be used to express the traversal from *Customer* to *Product*, corresponding to the composition of morphisms 22, 23, 35, and 39. (Note that we use two morphisms in the example, namely, $23.22 : Customer \rightarrow Order$ and $39.35 : Order \rightarrow Product$ to represent the same path.)

II. Query Decomposition The decomposition of a query into so-called *query parts* [1] 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 Figure 3.16, 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 [150].

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.

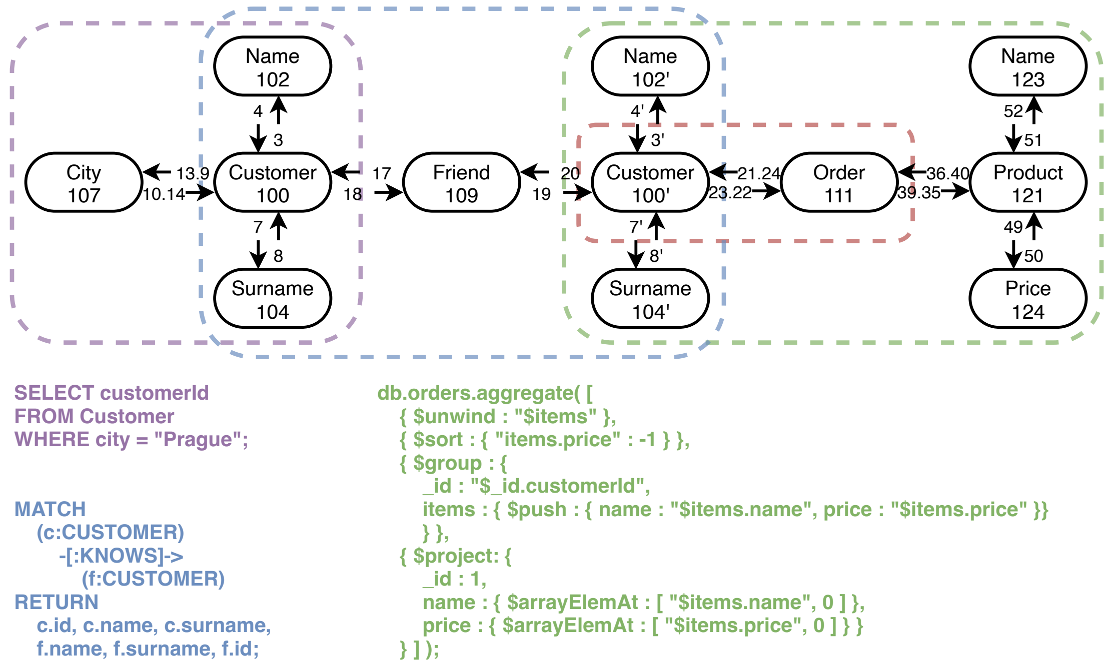


Figure 3.16: The decomposition of projection pattern and examples of translations of corresponding query parts into particular statements

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 utilise 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 respective indices. Similarly, the joining of aggregates can also be expensive due to possibly denormalised 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* [20, 146], i.e., a generalisation of the Cartesian square and intersection. As illustrated in Figure 3.17 using the respective colours, 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., $P_2 = P_1 \bowtie_{100'} result_{DOC}$).

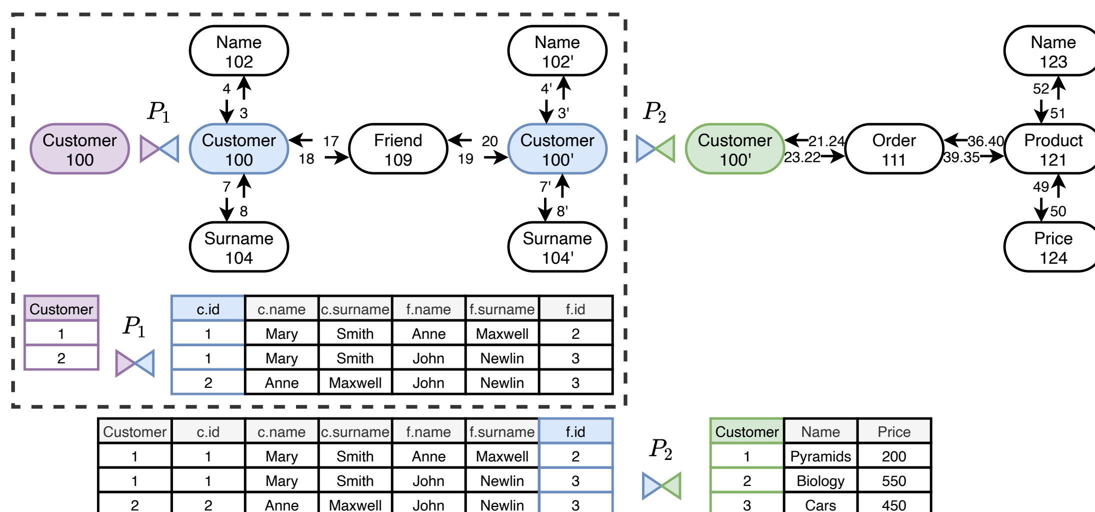


Figure 3.17: Joining of intermediate results by pullbacks

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 [113] 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 Figure 3.18.

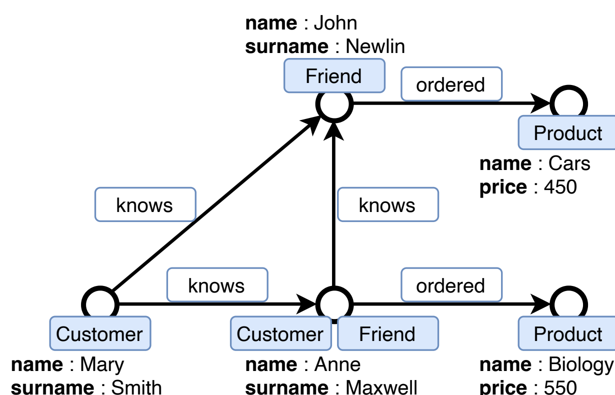


Figure 3.18: The result of the query represented in a graph model

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, 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.

3.7 Related Work

Each of the existing multi-model DBMSs [13] 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* [27] represents the data as named collections, each containing a set of blocks consisting of a non-empty set of entries. *Associative arrays* [28] are defined as mappings from pairs of unique (column and row) keys to values. Or, the *Tensor Data Model* [29] introduces the idea of generalised 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* [137] 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 [138], 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 [151, 152, 153]. Recently, paper [30] 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. 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 representa-

tion 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 [134, 20], or object-relational, i.e., hierarchies of classes [19]) and define a respective schema category and operations using standard categorical approaches (such as functors). Paper [135] 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 [18] defines a schema category covering various conceptual modelling 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.

3.8 Conclusion

In this paper, we continue to build a general framework for unified modelling 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 non-standardised multi-model world uniformly and precisely.

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

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 materialised 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 optimising 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* [149] or *SPARQL* [154] 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 [30]). 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.

3.8.1 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).

Paper IV

A Universal Approach for Multi-Model Schema Inference

Pavel Koupil[@1], Sebastián Hricko[1], Irena Holubová[1]

Accepted in Journal of Big Data by Springer Nature
doi: 10.1186/s40537-022-00645-9

[@] corresponding author, e-mail: pavel.koupil@matfyz.cuni.cz

[1] Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Abstract

The ***variety*** feature of Big Data, represented by *multi-model data*, has brought a new dimension of complexity to all aspects of data management. The need to process a set of distinct but interlinked data models is a challenging task.

In this paper, we focus on the problem of inference of a schema, i.e., the description of the structure of data. While several verified approaches exist in the single-model world, their application for multi-model data is not straightforward. We introduce an approach that ensures inference of a common schema of multi-model data capturing their specifics. It can infer local integrity constraints as well as intra- and inter-model references. Following the standard features of Big Data, it can cope with overlapping models, i.e., data redundancy, and it is designed to process efficiently significant amounts of data.

To the best of our knowledge, ours is the first approach addressing schema inference in the world of multi-model databases.

Keywords

• Multi-model data • Schema inference • Cross-model references • Data redundancy

4.1 Introduction

The knowledge of a schema, i.e., the structure of the data, is critical for its efficient processing. We can distinguish *schema-full*, *schema-less*, and *schema-mixed* database management systems (DBMSs), where the schema definition is required, ignored, or can be only partial. However, despite the specification of a schema when storing the data (i.e., the so-called *schema-on-write* approach) is not required in some systems, the knowledge of the structure of the data is needed when the data is to be processed, i.e., the so-called *schema-on-read* approach is still essential. Hence, when the user does not define the schema, it needs to be extracted from the data.

The problem of *inference of a schema* for a given data has been studied for several years, mainly for XML [71] and JSON [155], i.e., the document model, which has the richest structure among the current common models. For XML documents, where the order of elements is significant, the respective schemas involve regular expressions that describe the structure of the data. According to the Gold's theorem [87] regular languages are not identifiable only from positive examples (i.e., sample XML documents), so either heuristics [78, 156] or a restriction to an *identifiable* subclass of regular languages [84] is applied. Newer approaches for currently popular JSON format, where the order is not captured in the schemas and, thus, the inference process is in this manner less complex, focus mainly on schema inference for Big Data [65, 92]. However, the ***volume*** of Big Data is not its only challenge. The ***variety*** feature represented by the *multi-model data* adds a new dimension of complexity – the need to process a set of distinct but interlinked data models.

Example 4.1. Figure 4.1 provides an example of a scenario inspired by the multi-model benchmark *UniBench*.[1] It depicts an ER model where we omit attributes, identifiers, and cardinalities for the sake of simplicity.[2] The colours denote the particular logical models in which the respective part of the ER model is represented – blue graph, violet relational, yellow key/value, and two document models, green JSON and grey XML. The example represents an e-shop where customers, members of a social network capturing mutual acquaintance, order products from various vendors. □

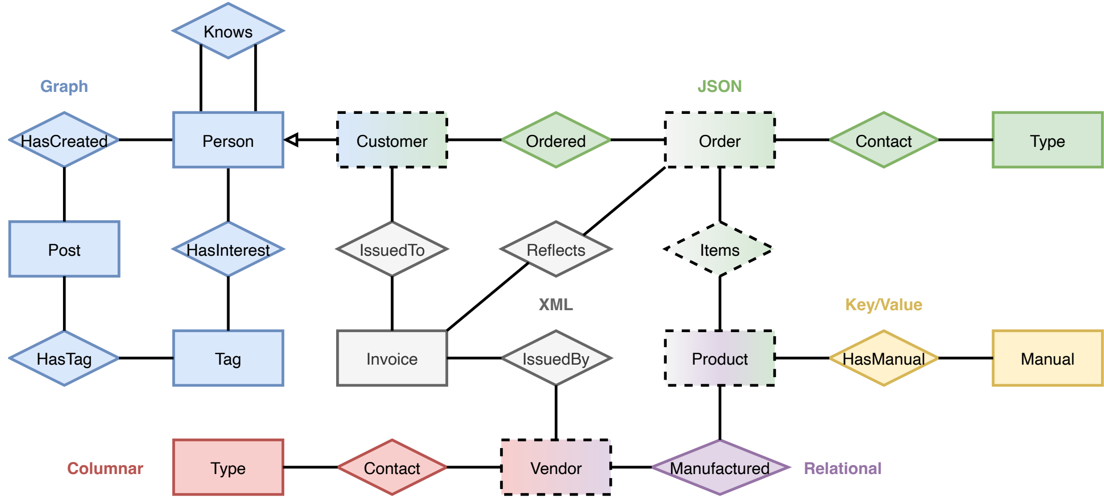


Figure 4.1: Extended *UniBench* multi-model scenario

At the logical level, the transition between two models can be expressed either via (1) *inter-model references* or by (2) *embedding* one model into another (such as, e.g., columns of type *JSONB* in relational tables of *PostgreSQL*[3]). Another possible combination of models is via (3) *cross-model redundancy*, i.e., storing the same data fragment in multiple models.

In the case of multi-model data, the problem of schema inference is further complicated by contradictory features of the combined models (such as structured vs semi-structured, aggregate-oriented vs aggregate-ignorant, order-preserving vs order ignoring etc.), inter-model references and cross-model integrity constraints (ICs) in general, the existence of a (partial) schema in schema-full/schema-mixed systems preserving the data, or cross-modal redundancy. Besides, there are verified single-model approaches that, however, naturally target only specifics of the particular data model. And, last but not least, the question is how to represent the resulting multi-model schema, i.e., whether to choose one of the models (and which one) or whether a more abstract representation, such as UML [32], is a better choice.

To address the key indicated problems, we extend our previous research results both in the area of inference of an XML schema [78, 156] and unified management of multi-model data [64, 2]. We propose a novel approach capable of inference of a schema for a given set of multi-model data. The main contributions are as follows:

[1] http://udbms.cs.helsinki.fi/?projects/ubench

[2] The full model will be provided in the following examples.

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

- In the proposed approach, we support all popular data models (relational, array, key/value, document, column, graph, and RDF) and all three types of their combination (embedding, references, and redundancy).
- We can cover schema-less, schema-mixed, and schema-full systems, i.e., if needed, we can re-use an existing schema both user-defined or inferred using a verified single-model approach.
- We support both local integrity constraints (e.g., unique or primary key) and global integrity constraints, i.e., intra-model and inter-model references.
- We introduce two versions of the approach – record-based and property-based – and experimentally verify their appropriateness for structurally different data.
- Following the current trends, the approach is designed to be parallelisable and, thus, scalable for Big Data.
- The proposed approach was implemented as a tool called *MM-infer* [109],[4] i.e., the proof of the proposed concept.

Outline The rest of the paper is structured as follows: In Section 4.2 we overview related work and motivate the proposed approach. In Section 4.3 we discuss the currently popular data models, their specifics, and the respective influence on schema inference. In Section 4.4 we describe in detail the proposed approach. Section 4.5 describes the architecture and implementation details of *MM-infer* and Section 4.6 introduces results of experiments. In Section 4.7 we conclude and outline future work.

4.2 Related Work

Several papers currently deal with the inference of a schema for a given set of sample data. We can divide them into approaches inferring (1) structural and (2) semantic schema. The approaches focus mainly on the document model expressed using XML or JSON in the former case. The critical difference is whether the order of child properties is significant or not. And in addition, since the JSON documents are closely related to NoSQL databases and Big Data, the approaches often support scalable processing, i.e., they can be parallelised. In the latter case of inference of a semantic schema, the aim is different. The approaches focus on the inference of a schema describing the semantics of the information stored in the data, usually expressed in RDF [66], but not its logical structure within a selected data model. Since this is not our current main target, we refer an interested reader to a recent extensive survey in [157].

XML Schema Inference An extensive comparison of XML schema inference approaches can be found in [72]. The approaches are older, reflecting the decreasing popularity of XML with the arrival of Big Data and JSON. They can

[4] https://www.ksi.mff.cuni.cz/~koupil/mm-infer/

be classified according to various criteria such as, e.g., the type of the result (i.e., the language used), the way it is constructed, the inputs used, etc.

Heuristic approaches [74, 75, 76, 77, 78, 79] are based on experience with manual construction of schemas. Their results do not belong to any specific class of XML grammars, and they are based on the generalisation of a trivial schema using a set of predefined heuristic rules, such as, e.g., "if there are more than three occurrences of an element, it is probable that it can occur arbitrary times". These techniques can be further divided into methods that generalise the trivial schema until a satisfactory solution is reached [75, 76, 78] and methods that generate a considerable number of candidates and then choose the best one [77]. While in the first case, the methods are threatened by a wrong step which can cause the generation of a suboptimal schema. In the latter case, they have to cope with space overhead and specify a proper function for the evaluation quality of the candidates. A special type of heuristic methods are so-called *merging state algorithms* [76, 78]. They are based on the idea of searching a space of all possible generalisations, i.e., XML schemas, of the given XML documents represented using a prefix tree automaton. By merging its states and thus generalising the automaton, they construct the sub-optimal solution. Since the space is theoretically infinite, only a proper subspace of possible solutions is searched using various heuristics.

On the other hand, methods based on *inferring of a grammar* [80, 81, 82, 83, 84, 85, 86] exploit the theory of languages and grammars and thus ensure a certain degree of quality of the result. We can view an XML schema as grammar and an XML document valid against the schema as a word generated by the grammar. Although grammars accepting XML documents are, in general, context-free [158], the problem can be reduced to inferring a set of regular expressions, each for a single element (and its subelements). But, since, according to Gold's theorem [87] regular languages are not identifiable only from positive examples (i.e., sample XML documents which should conform to the resulting schema), the existing methods exploit various other information such as, e.g., the predefined maximum number of nodes of the target automaton, restriction to an identifiable subclass of regular languages, etc.

JSON Schema Inference The current popular JSON schema inference approaches are described and compared in [16]. Paper [90] statically compares several schema extraction algorithms over multiple NoSQL stores.

Paper [92] presents an approach for inferring versioned schemas from document NoSQL databases based on the *Model-Driven Engineering* (MDE) along with sample applications created from such inferred schemas. This research is furthered by dissertation thesis [90] and by paper [93] who tackle the issues of visualisation of schemas of *aggregate-oriented* NoSQL databases and propose desired features that should be supported in visualisation tools. Most recently, Fernandez et al. expand upon the meta-model from paper [92] by introducing a unified meta-model capable of modelling both NoSQL and relational data [30].

Authors of [94] propose an approach to extract a schema from JSON data stores, measuring the degree of heterogeneity in the data and detecting structural outliers. They also introduce an approach for reconstructing schema evolution history of *data lakes* [95]. Additionally, *jHound* [96], a JSON data profiling tool

is presented, which can be used to report key characteristics of a dataset, find structural outliers, or detect documents violating best practices of data modelling. Finally, *Josch* [97] is a tool that enables NoSQL database maintainers to extract a schema from JSON data more efficiently, refactor it, and then validate it against the original dataset.

Authors of [65] propose a distributed approach for parameterized schema inference of massive JSON datasets and introduce a simple but expressive JSON type language to represent the schema.

Paper [98] provides an MDE-based approach for discovering schema of multiple JSON web-based services. Later its authors put it into practice as a web-based application along with a visualisation tool [99].

Last but not least, Frozza et al. introduce a graph-based approach for schema extraction of JSON and BSON[5] document collections [89] and another inference process for columnar NoSQL databases [103], specifically HBase.[6]

Summary As we can see, the amount of related work is significant, and there exist approaches focusing on many specifics of schema inference. However, to the best of our knowledge, currently, there exists no approach that deals with the inference of a multi-model schema. At first sight, the single-model approaches can be reused. However, this idea is not that straightforward. As we will show in the following sections, the particular models have distinct, even contradictory features, so first, a way to unify them must be found. Another complication is the mutual references and redundancy that need to be considered.

4.3 Data Models and Their Unification

In the rest of our work, we consider the following currently popular data models: relational, array, key/value, document, column, graph, and RDF, i.e., we support all currently popular structured and semi-structured data to cover all combinations of models used in the existing popular multi-model systems.[7] First, we provide a brief overview of their features. Next, we discuss their unification to simplify and clarify the further explanation of the proposal.

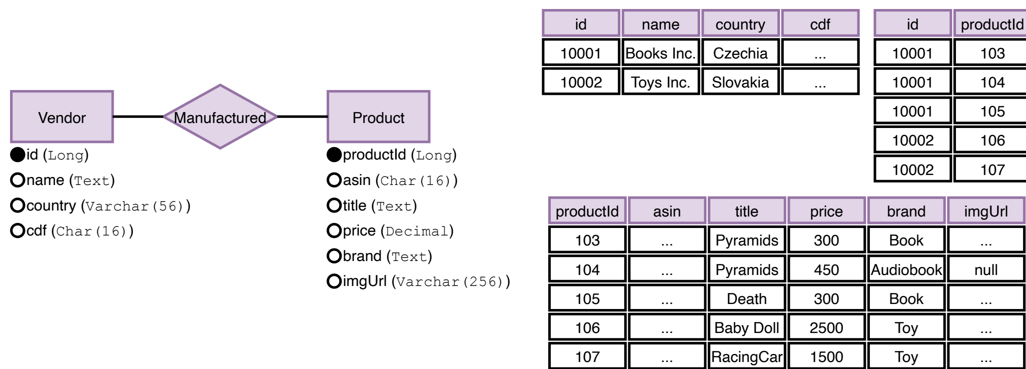
4.3.1 Overview of Models

From the structural point of view, which is our main target, the core classification is based on the complexity of the supported data structures. *Aggregate-oriented models* (key/value, document, and column) 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 data de-normalisation. On the contrary, *aggregate-ignorant models* (relational, array, graph, and RDF) are not primarily oriented to the support of aggregates. The relational world strongly emphasises the normalisation of structured data, whereas the graph model is, in principle, a set of flat objects mutually linked by any number of edges.

[5] https://bsonspec.org
[6] http://hbase.apache.org
[7] https://db-engines.com/en/ranking

Relational Model Relational model is based on the mathematical term *relation*, i.e. a subset of Cartesian product. The data are logically represented as tuples forming relations. Each tuple in a relation is uniquely identified by a *key*. A part of the *Structured Query Language* (SQL) [159], called *Data Definition Language* (DDL), is denoted for the definition of a relational schema, i.e., the names of relation, names of attributes, their domains (simple data types), and integrity constraints (i.e., keys, foreign keys etc.).

Example 4.2. In Figure 4.2 we can see an example of data from the relational model, namely the one implemented in *PostgreSQL* as reflected by the particular data types. On the left we can the respective part of ER model from Figure 4.1 and its transformation to three respective tables (relations) *Vendor*, *Product*, and *Manufactured* with the respective columns. □



id	name	country	cdf
10001	Books Inc.	Czechia	...
10002	Toys Inc.	Slovakia	...

id	productId
10001	103
10001	104
10001	105
10002	106
10002	107

productId	asin	title	price	brand	imgUrl
103	...	Pyramids	300	Book	...
104	...	Pyramids	450	Audiobook	null
105	...	Death	300	Book	...
106	...	Baby Doll	2500	Toy	...
107	...	RacingCar	1500	Toy	...

Figure 4.2: An example of relational data

Array Model The array model works with the notion of *multi-dimensional array* being represented as a mapping from a set of dimensions to a set of attributes. In this sense, the relational model represents the case of one dimension, i.e., the identifier (index) of a particular tuple of a relation, or two dimensions corresponding to an identifier of a tuple and a particular attribute. Also, in this case, the DDL specifies the structure of the arrays, i.e., their names, the domains, ranges, and steps of dimensions, the names and domains of attributes, and respective integrity constraints.

Example 4.3. In Figure 4.3 we can see how the the same part of ER schema used in Figure 4.2 would be transformed to the array model. While one-dimensional arrays *Vendor* and *Product* have the same structure, two-dimensional array *Manufactures* occupies much more space. □

Key/Value Model The key/value model is the simplest aggregate-oriented data model. It corresponds to associative arrays, dictionaries, or hashes. Each record in the key/value model consists of an arbitrary schema-less value and its unique key, enabling storing, retrieving, or modifying the value.

Example 4.4. In Figure 4.4 we can see an example of key/value data. Both the ER model on the left and visualisation of the data on the right depict the simplicity of the model – the identifier *ProductID* and respective unstructured (binary) data content denoted as *Content*. □

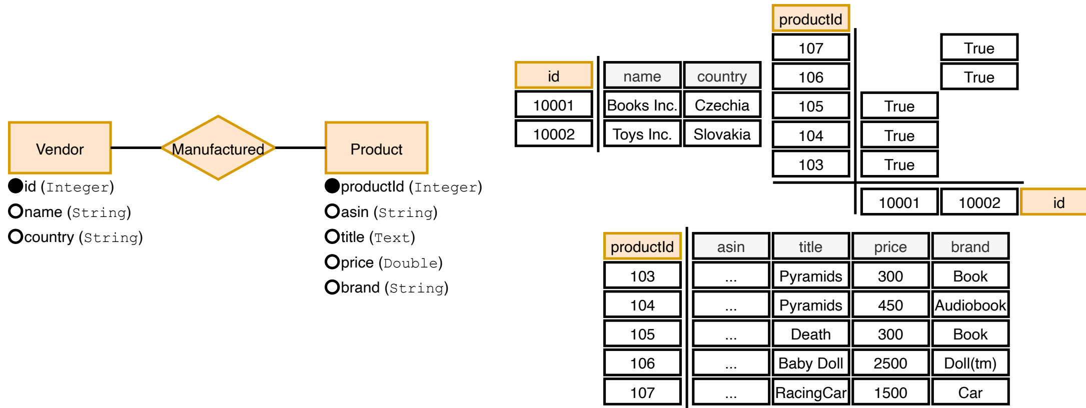


Figure 4.3: An example of array data

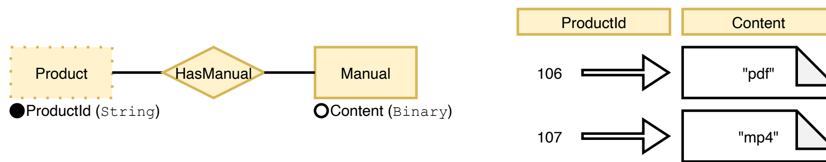


Figure 4.4: An example of key/value data

Column Model The (wide) column model can be interpreted as a two-dimensional key/value model. It consists of the notions of a column family (table), a row, and a column. However, unlike the relational model, each row of a column family (table) can have different columns (having different names and/or data types). In other words, each row is a set of key/value pairs independent of other rows of the same column family. In some wide column systems, such as *Cassandra*,[8] it is possible to specify (a part of) a schema of column families. Usually, a set of optional/compulsory columns is common for rows of the column family, whereas others can be arbitrary. If only a part of the schema can be specified, we speak about schema-mixed systems.

Example 4.5. In Figure 4.5, we can see sample column data, namely the approach used in *Cassandra*, corresponding to the respective part of ER model on the left. Each row in the column family on the right is identified using column *id* and further contains three columns *name*, *country*, and *cdf* of type *String*. Next, it contains column *Industry* of complex type *Set<String>* (i.e., a set of strings) and column *Contact* of type *Map* which is supported by *Cassandra*. Since the column *Industry* is not compulsory, the respective value is missing in some rows. □

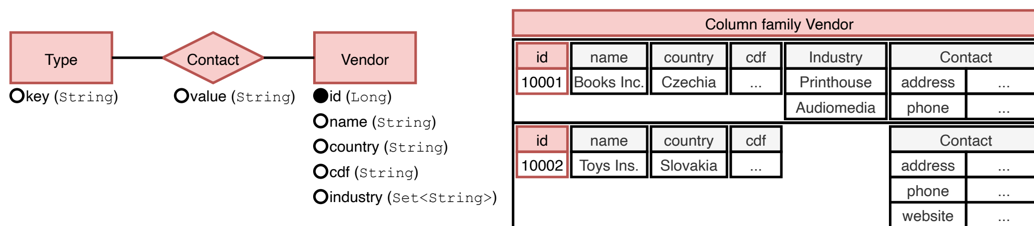


Figure 4.5: An example of column data

[8]https://cassandra.apache.org/_/index.html

Document Model The document (semi-structured) model is based on the idea of representing the data without an explicit and separate definition of its schema. Instead, the particular pieces of information are interleaved with structural/semantic tags that define their structure, nesting etc.

The XML is a human-readable and machine-readable markup language. The data are expressed using elements delimited by tags containing simple text, subelements, or their combination. Additional information can be stored in the attributes of an element. Standard languages like *Document Type Definition* (DTD) [71] or *XML Schema* [104, 105] enable to specify the structure of XML documents using regular expressions.

Example 4.6. Figure 4.6 represents an example of document data expressed in XML. As we can see, the structure of invoices can differ depending on whether the invoice is for a customer or for a vendor. The XML document with root element *invoice* identified by attribute *invoiceNo* contains identification of the respective customer (element *customerId*) or vendor (element *vendorId*), date of creation (element *creationDate*), due date (optional element *dueDate* present if the invoice is not paid yet), list of ordered items (subelements *product* of element *items*), total price of the order (element *totalPrice*), and an indication whether it was already paid (optional element *paid*). So, there can exist several structurally different version of an invoice in the document collection depending on their status. □

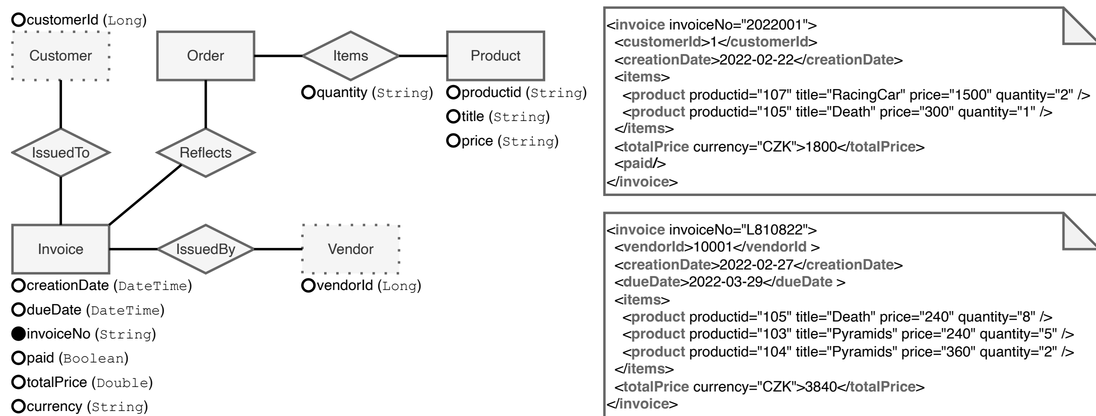


Figure 4.6: An example of document data expressed in the XML format

The JSON is a human-readable open-standard format. It is based on the idea of an arbitrary combination of three basic data types used in most programming languages – key/value pairs, arrays, and objects. Contrary to XML, the specific order of items in a JSON document is not essential. *JSON Schema* [106] language enables to specify the structure of JSON documents.

Example 4.7. In Figure 4.7 we can see an example of document data expressed in JSON. Again, we can see two structurally different documents belonging to the same collection, this time describing orders. The order is identified using a simple property *_id* and further contains embedded documents *customer*, *contact*, and *Items*. Note that some information about customers and products may be stored redundantly in each order where the customer or product appears. In addition, since the property *contact* represents a map which is not supported in JSON, it corresponds to a set of optional properties. □

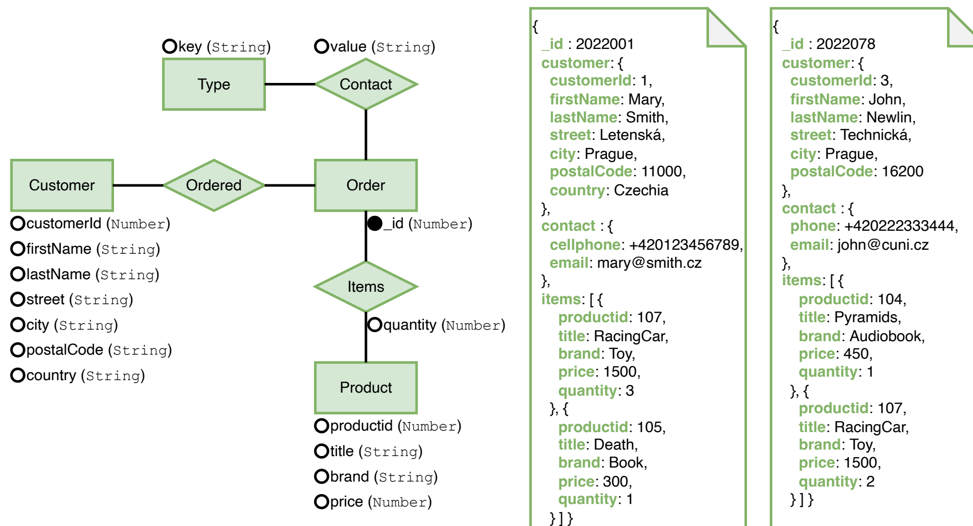


Figure 4.7: An example of document data expressed in the JSON format

Graph Model The graph data model is based on the mathematical definition of a *graph*, i.e., a set of vertices (nodes) V and edges E corresponding to pairs of vertices from V. Nodes and edges are assigned with attributes, each having a name and domain (simple type). In addition, both nodes and edges have their type, enabling to group nodes/edges that represent the same piece of reality. However, the schema of nodes/edges does not (or even is not expected to) be defined.

Example 4.8. Figure 4.8 provides an example of graph data. As we can see, from the structural aspect, the model contains only nodes, edges, and attributes with simple types. In this example, the nodes correspond to ER entities (i.e., *Post*, *Tag*, *Person*, and *Customer*) and edges to respective ER relationships. □

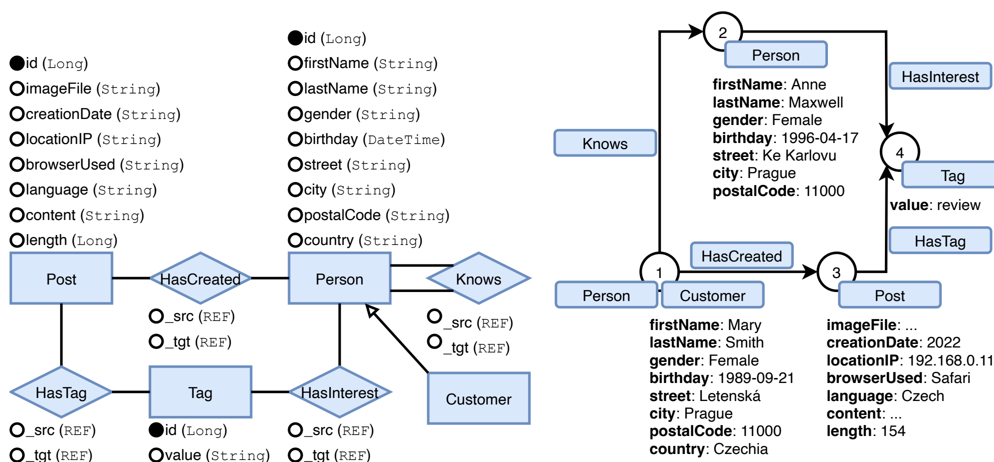


Figure 4.8: An example of graph data

RDF Model The RDF model corresponds to a directed graph composed of triple statements. The statement is represented by a node for the subject, a node for the object, and an edge that goes from the subject to the object represent-

ing their mutual relationship. Each of the three parts of the statement can be identified by a URI.

The *RDF Schema* (RDFS) [160] or the *Web Ontology Language* (OWL) [161] enable to define a schema of RDF data. However, it does not express the structure of the data, but the semantics, i.e., classes to which the represented entities of the real word belong, their features and mutual relationships etc.

Multi-Model Data Multi-model data are in general data which are logically represented in more than one data model. The currently existing multi-model DBMSs [13] differ in the strategy used to extend the original model to other models or to combine multiple models. The new models can be supported by adopting an entirely new storage strategy, an extension of the original storage strategy, a new interface, or even no change in the original storage strategy (used for trivial cases). From the logical level, the transition between two models can be expressed either via:

1. *inter-model references*,
2. *embedding* one model into another (e.g., columns of type JSON in tables of the relational model of *PostgreSQL*[9]), or
3. *multi-model redundancy*, i.e., storing the same data fragment in two or more distinct models, usually for efficient query evaluation.

Example 4.9. For the sake of clarity, the complete set of sample multi-model data corresponding to the ER model in Figure 4.1 is provided in Figure 4.9. The figure illustrates the need for a multi-model schema inference approach. As we can see, even this simple example depicts how hard it is to manage multi-model data and discover the overall structure, references, redundancy etc., within distinct models and their specifics. □

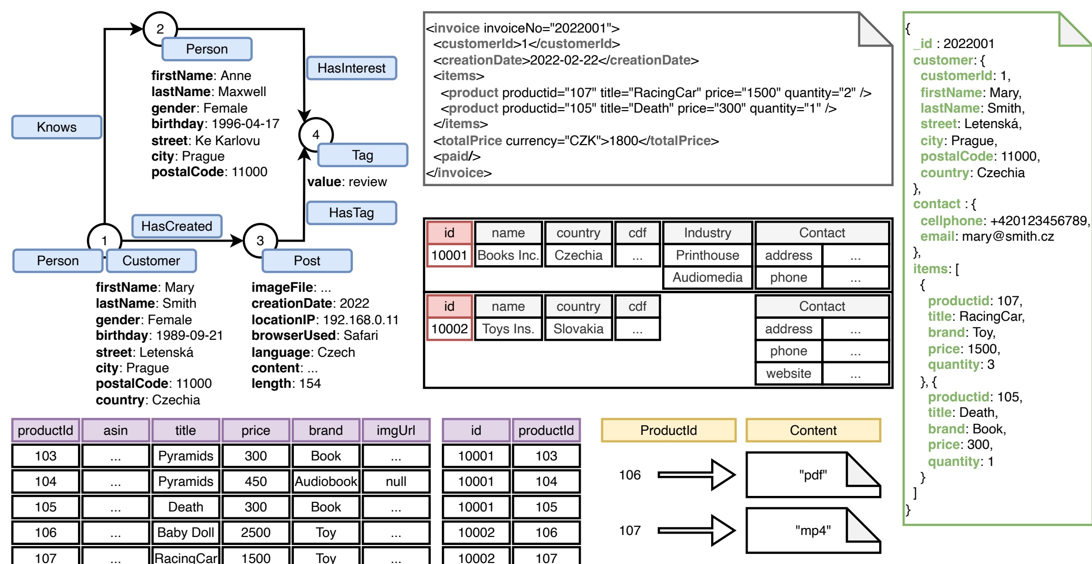


Figure 4.9: An example of multi-model data

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

Regarding the definition of a schema, currently there exists no standard language for expressing the structure of multi-model data. More abstract approaches, such as the *Entity-Relationship (ER) model* [31, 147] or the *Unified Modeling Language* (UML) [32], or our proposal [2] based on category theory [133] can be used.

4.3.2 Unification of Models

Since the terminology within the considered models differs, we provide a unification used throughout the text in Table 4.1.

Table 4.1: 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	JSON Field / XML element or attribute	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 / dimensions	Identifier	Subject	Key	JSON identifier / XML ID or key	Row key
Reference	Foreign key	–	–	–	–	JSON reference / XML keyref	–
Array	–	–	Array	–	Array	JSON array / repeating XML elements	Array
Structure	–	–	–	–	Set / ZSet / Hash	Nested document	Super column

As we can see, the terminology is apparent in most cases, but some comments might be needed in specific cases. A *kind* represents a single logical class of items represented in each of the models. For instance, in the relational model, a kind corresponds to the notion of a table, whereas in the graph model, a kind corresponds to a class of nodes/labels specified by a label. A *record* then represents one particular item of a kind, e.g., a row of a table or a graph node/edge with a particular label.

A record consists of *properties* which can be either simple, i.e., having a simple scalar value, or complex, i.e., containing other properties. The complex properties enable hierarchically nested structure in the case of the document model and also the column model in some cases (e.g., in *Cassandra*, where super-columns, i.e., a two-level hierarchy of the columns, are supported). With this view, the whole

kind can be treated as a single complex *top-level property*, whose name is the name of the kind and its child properties correspond to the properties of the kind. For example, in the case of the document model, there is only one such child property – the root element of the XML document or an anonymous root of the JSON document. Or, in the case of the relational model, the child properties of the top-level property correspond to the particular columns. We will use this view to simplify the inference algorithms.

Domains and *values* correspond to data types and selected values in all the models. *Identifiers* correspond to the notion unambiguously identifying particular records. *References* from one kind to another are allowed only in the relational and document model.

Last but not least, considering more complex data types, some models support *arrays* or *structures*. 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., represent nested documents); in all other cases, only arrays of simple (scalar) types are allowed.

4.4 Multi-Model Schema Inference

We have to assume that the input data may be stored either in one multi-model DBMS or in a *polystore*, i.e., a set of single-model or multi-model systems. To infer a multi-model schema, we first need to process all types of input data. In addition, for some systems, a (partial) user-defined schema may exist. Or, for some models, a verified single-model inference approach may exist. These results can be integrated into the multi-model result.

First, in Section 4.4.1 we introduce two basic building blocks of the proposed approach:

1. a *type hierarchy* used for inference of data types and
2. a unifying representation of a single record or a possibly existing schema called *Record Schema Description*.

Next, we provide an overview of the general workflow of the algorithm in Section 4.4.2. Then, we introduce in detail the inference approach, namely its record-based (Section 4.4.3) and property-based (Section 4.4.4) version. Finally, we discuss the process of inference of integrity constraints (Section 4.4.5).

4.4.1 Basic Building Blocks

To be able to process all the considered data models using a single unified approach, we propose two auxiliary structures – the type hierarchy (see Section 4.4.1) and the record schema description (see Section 4.4.1). The *type hierarchy* enables us to cope with the problem of distinct sets of simple types used in the models (or their system-specific implementations). The *record schema description* enables one to describe a schema of one kind (including the case of a schema of a single record) regardless of its model(s).

Type Hierarchy

The *type hierarchy* is a simple tree-based representation of basic data types that can occur in the input models and their mutual relationships. It enables us to quickly find a common supertype that covers a set of data types even though not all of them are supported in each model.

The data types supported across the data models, forming the set $\mathbb{T}'$, are represented as vertices v_i and the natural hierarchy between the types is represented as edges $e : v_j \rightarrow v_i$, when values of v_j involve also values of v_i. For example, the fact that `String` is a generalisation of `Integer` is represented as `String` $\rightarrow$ `Integer`. Additionally, we assign a unique prime p_i or number 1 to each vertex v_i. The numbers are assigned using the BFS algorithm, starting from 1 assigned to the root of the hierarchy and ensuring that $p_i < p_j$ if $e : v_i \rightarrow v_j$. The integer representation of node v_i is then computed as $T_i = \prod_{j=0}^{i} p_j$, where $p_0, \ldots, p_i$ are prime numbers (or 1 assigned to the root) assigned to vertices $v_0, \ldots, v_i$ on the path from root node v_0 to v_i. The concept of *union type* UT, is recursively defined as a union (denoted using operator $\oplus$) of types $T_i, i = 0, ..., n$, i.e., $UT := \oplus_{i=0}^{n} T_i$, where T_i is a simple type or a union type. We will denote the set of all types, i.e., both basic and union types as $\mathbb{T}$.

Next, we can introduce an associative and distributive operator *bestGeneralType*, defined as $T_i \sqcup T_j := gcd(T_i, T_j)$, where gcd is the greatest common divisor of prime products T_i and T_j representing the best general type of T_i and T_j.

In addition, we introduce additional associative and distributive operator *typeMerge*, denoted as $\diamond$, defined as:

- $T_i \diamond T_j := T_i \oplus T_j$ if $T_i \neq T_j$
- $UT \diamond T := UT \oplus T$ if $T \nsubseteq UT$
- $UT \diamond T := UT$ if $T \subseteq UT$
- $UT_1 \diamond UT_2 := UT_1 \oplus UT_2$ if $UT_1 \nsubseteq UT_2$
- $UT_1 \diamond UT_2 := UT_1$ if $UT_2 \subseteq UT_1$

Example 4.10. Let us illustrate an example of the type hierarchy of data types used in Figure 4.1. The hierarchy illustrated in Figure 4.10 is represented as a tree having an artificial root *AnyType*[10] and then the natural type hierarchy. As we can see, we do not consider the hierarchy typical for programming languages, i.e., an object being a supertype. Instead, we consider representation hierarchy, i.e., natural conversions between data types. For example, since anything can be represented as a *String*, it is the root of the tree. Additionally, *String* is assigned with 1, i.e., *String* is a supertype of all types.

As we can also see, each node is assigned with prime (in the BFS order), whereas the data type itself is represented as a product (corresponding to the path from *String* to the data type). For example, $String := 1$, $Tuple := 1 \times 2 \times 19 \times 43$, and $Double := 1 \times 11 \times 31$.

Additionally, in the example we also represent any *ComplexType* as an even number, because $Collection := 1 \times 2$ is a supertype of all complex types. This

[10] In usual implementations, we consider *String* as a supertype of all data types, therefore the numbering of nodes starts from *String* instead of *AnyType* (i.e., *AnyType* is ignored).

allows us to quickly distinguish simple and complex types using binary operations (lowest bit value test). □

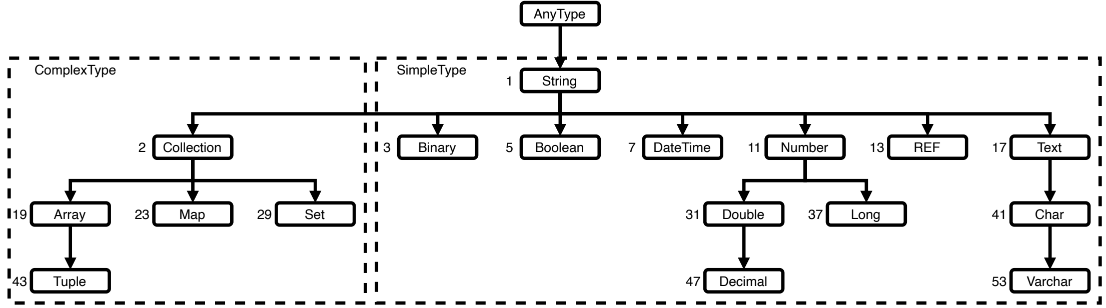


Figure 4.10: An example of Type Hierarchy

Example 4.11. Having the data type hierarchy from Figure 4.10, we can represent a union type as a union of data types represented as products. For example:

- A union type $UT_s := Double \oplus Long$ of two simple types is computed as a simple union of products, i.e., $UT_s := (1 \times 11 \times 31) \oplus (1 \times 11 \times 37)$.[11]

- A union of two complex types $UT_c := Tuple \oplus Map$ is represented as $UT_c := 2 \times 19 \times 43 \oplus 2 \times 23$.

- Finally, a union of two union types $UT_u := UT_s \oplus UT_c$ is represented as a union $UT_u := 11 \times 31 \oplus 11 \times 37 \oplus 2 \times 19 \times 43 \oplus 2 \times 23$.

□

Example 4.12. Computing the best general type can be done as follows:

- Having a union type of simple types $UT_s := 11 \times 31 \oplus 11 \times 37$, the best best general type is computed as $gcd(11 \times 31, 11 \times 37) = 11 \cong Number$.

- Having $UT_c := 2 \times 19 \times 43 \oplus 2 \times 23$, the best general type is computed as $gcd(2 \times 19 \times 43, 2 \times 23) = 2 \cong Collection$.

- Finally, the best general type of $UT_u := 11 \times 31 \oplus 11 \times 37 \oplus 2 \times 19 \times 43 \oplus 2 \times 23$ is $gcd(11 \times 31, 11 \times 37, 2 \times 19 \times 43, 2 \times 23) = 1 \cong String$.

□

Note that the implementation of finding the best general type may follow the hierarchical structure of the tree. Therefore it is not needed to compute *gcd*() in an explicit way (with the exponential complexity of the algorithm). Instead, we traverse the tree from its root, and we try to divide all the type representations by a prime assigned to the node. If it returns an integer, we traverse deeper. Otherwise, we try the sibling nodes. Having all the siblings processed and no subtree to traverse, the *gcd* is found.

[11]Note that parentheses can be omitted with regards to the order of operations $\times$ and $\oplus$. Also $String := 1$ can be omitted in the product.

Record Schema Description

The *Record Schema Description* (RSD) enables us to describe a schema of one kind regardless of its model(s). It naturally covers also the case of a *trivial schema* of a single record. So, in the proposed multi-model inference process it serves for representation of:

1. All types of *input schemas*, i.e.,
   (a) an existing user-defined schema,
   (b) a single-model schema inferred using a single-model approach, and
   (c) a *basic schema* inferred using the *local schema inferrer* in the remaining cases, i.e., for kinds without a schema,
2. *Intermediate schemas* created during the inference process by the *global schema inferrer*, and
3. The *resulting multi-model schema* that is transformed to a required output form.

The RSD has a tree structure, and it describes the structure of a property, a record, or a kind (i.e., a set of records) because all the three cases can be treated in the same way, as we have discussed above. The root of an RSD corresponds to a root property of the respective data model (e.g., the root XML element or the anonymous root of a JSON hierarchy), or it is an artificial root property with trivial settings encapsulating the properties (e.g., in the relational or graph model). An RSD describing a property (or a record or a kind) p is recursively defined as a tuple rsd = ($name$, $unique$, $share$, id, $types$, $models$, $children$, $regexp$, ref), where:

- $name$ is the name of property p extracted from the data (e.g., `_id`, `person`) or it can be anonymous (e.g., in case of items of JSON arrays or an artificial root property).
- $unique$ is the IC specifying uniqueness of values of p. Its values can be `T` (true), `F` (false), or `U` (unknown) for intermediate steps of the inference process.
- $share = (share_p, share_x)$ is a tuple, where $share_p$ is the number of all occurrences of property p and $share_x$ is the number of parent properties containing p at least once. Note that $share_p > share_x$ reflects so-called *repeating* property, i.e., property forming an element of an array. Also note that if we combine $p.share_x$ with $pp.share_p$ (of any type), where pp is the parent property of p, we get the optionality of p, i.e., $p.share_x = pp.share_p$ reflects a required property, while $p.share_x < pp.share_p$ reflects an optional property.
- id is the IC specifying that the property is an identifier. Its values can also be `T`, `F`, or `U` with the same meaning.[12]

[12] For the sake of simplicity we currently do not support composite identifiers or respective composite references.

- *types* is a set of data types that cover the property. For a simple property it involves simple data types (i.e., `String`, `Integer`, ...). For a complex property it involves the following values:
  - `Array`, i.e., ordered (un)named (not) unique child properties (e.g., child elements in XML or items of arrays in JSON),
  - `Set`, i.e., unordered unnamed unique child properties (e.g., items of `Set` in column store *Cassandra*), and
  - `Map`, i.e., unordered named unique child properties (e.g., attributes of a relational table).

  As we will see later, in the final phase of the inference process, the set is reduced to a single resulting datatype.
- *models* is a (possibly empty) set of models (`JSON` = JSON document, `XML` = XML document, `REL` = relational, `GRAPH` = graph, `COL` = column, `KV` = key/value) that involve the property. If the set contains more than one item, it represents cross-model redundancy. If the value of *models* within a child property changes, it corresponds to embedding one model to another.
- *children* is a (possibly empty) set of recursively defined child properties.
- (Optional) *regexp* specifies a regular expression over the set *children*, or its subset (e.g., in case of schema-mixed systems or case of XML elements and attributes, forming together child properties).
- (Optional) *ref* specifies that the property references another property (of another kind). Since we do not specify any restriction on the referencing and referenced property models, we also cover self-references and the third possible combination of multiple models, i.e., inter-model references.

Example 4.13. Figure 4.11 provides sample RSDs of kinds from Figure 4.1 (having the respective colours) in their textual form. Each node is described as a tuple of values of its above-listed components (in the given order) in curly brackets. If the set *children* is not empty, in curly brackets, there occur the child properties described in the same way. □

Having the unifying representation of all possible types of input data having any of the supported models, we can propose a much more straightforward multi-model inference approach. It is based on an essential feature of RSDs – the fact that two RSDs can be merged to describe a common schema of the respective kinds. The merging strategy is a part of the proposed approach (see Section 7).

4.4.2 Schema Inference Workflow

The proposed inference process takes into account the following features and specifics of the multi-model environment:

- various aspects of the combined data models and their specifics known for popular multi-model DBMSs [13] (such as sets / maps / arrays / tuples, (un)ordered properties, various treatments of missing values etc.),

relational table Product

```
( _, U, (5,5), U, Map, REL, {
  ( asin, F, (5,5), F, Char(16), REL, ε, ε, ε ),
  ( brand, F, (5,5), F, Text, REL, ε, ε, ε ),
  ( imgUrl, F, (4,4), F, Varchar(256), REL, ε, ε, ε ),
  ( price, F, (5,5), F, Decimal, REL, ε, ε, ε ),
  ( productId, T, (5,5), T, Long, REL, ε, ε, ε ),
  ( title, F, (5,5), F, Text, REL, ε, ε, ε )
}, ε, ε )
```

relational table Vendor

```
( _, U, (2,2), U, Map, REL, {
  ( cdf, F, (2,2), F, Char(16), REL, ε, ε, ε ),
  ( country, F, (2,2), F, Varchar(64), REL, ε, ε, ε ),
  ( id, T, (2,2), T, Long, REL, ε, ε, ε ),
  ( name, F, (2,2), F, Text, REL, ε, ε, ε )
}, ε, ε )
```

graph labels Person, Customer

```
( _, U, (2,2), U, Map, GRAPH, {
  ( birthday, F, (2,2), F, DateTime, GRAPH, ε, ε, ε ),
  ( city, F, (2,2), F, String, GRAPH, ε, ε, ε ),
  ( country, F, (1,1), F, String, GRAPH, ε, ε, ε ),
  ( firstName, F, (2,2), F, String, GRAPH, ε, ε, ε ),
  ( gender, F, (2,2), F, String, GRAPH, ε, ε, ε ),
  ( id, T, (2,2), T, Long, GRAPH, ε, ε, ε ),
  ( lastName, F, (2,2), F, String, GRAPH, ε, ε, ε ),
  ( postalCode, F, (2,2), F, String, GRAPH, ε, ε, ε ),
  ( street, F, (2,2), F, String, GRAPH, ε, ε, ε )
}, ε, ε )
```

graph label Knows

```
( _, U, (1,1), U, Map, GRAPH, {
  ( _src, F, (1,1), F, REF, GRAPH, ε, ε, Person(id) ),
  ( _tgt, F, (1,1), F, REF, GRAPH, ε, ε, Person(id) )
}, ε, ε )
```

column family Vendor

```
( _, U, (2,2), U, Map, COL, {
  ( cdf, F, (2,2), F, String, COL, ε, ε, ε ),
  ( contact, F, (2,2), F, Map, COL, {
    ( address, F, (2,2), F, String, COL, ε, ε, ε ),
    ( phone, F, (2,2), F, String, COL, ε, ε, ε ),
    ( website, F, (1,1), F, String, COL, ε, ε, ε )
  }, ε, ε ),
  ( country, F, (2,2), F, String, COL, ε, ε, ε ),
  ( id, T, (2,2), T, Long, COL, ε, ε, ε ),
  ( industry, F, (1,1), F, Set, COL, {
    ( _, F, (2,1), F, String, COL, ε, ε, ε )
  }, (_+), ε ),
  ( name, F, (2,2), F, String, COL, ε, ε, ε )
}, ε, ε )
```

key/value pair Feedback

```
( _, U, (2,2), U, Map, KV, {
  ( content, F, (2,2), T, Binary, KV, ε, ε, ε ),
  ( productId, T, (2,2), T, String, KV, ε, ε, ε )
}, ε, ε )
```

JSON document Order

```
( _, U, (2,2), U, Map, JSON, {
  ( _id, T, (2,2), T, Number, JSON, ε, ε, ε ),
  ( contact, F, (2,2), F, Map, JSON, {
    ( cellphone, F, (1,1), F, String, JSON, ε, ε, ε ),
    ( email, F, (2,2), F, String, JSON, ε, ε, ε ),
    ( phone, F, (1,1), F, String, JSON, ε, ε, ε )
  }, ε, ε ),
  ( customer, F, (2,2), F, Map, JSON, {
    ( city, F, (2,2), F, String, JSON, ε, ε, ε ),
    ( country, F, (2,2), F, String, JSON, ε, ε, ε ),
    ( customerId, F, (2,2), F, Number, JSON, ε, ε, ε ),
    ( firstName, F, (2,2), F, String, JSON, ε, ε, ε ),
    ( lastName, F, (2,2), F, String, JSON, ε, ε, ε ),
    ( postalCode, F, (2,2), F, String, JSON, ε, ε, ε ),
    ( Street, F, (2,2), F, String, JSON, ε, ε, ε ),
  }, ε, ε ),
  ( items, F, (2,2), F, Array, JSON, {
    ( _, F, (4,2), F, Map, JSON, {
      ( brand, F, (4,4), F, String, JSON, ε, ε, ε ),
      ( price, F, (4,4), F, Number, JSON, ε, ε, ε ),
      ( productId, F, (4,4), F, Number, JSON, ε, ε, ε ),
      ( quantity, F, (4,4), F, Number, JSON, ε, ε, ε ),
      ( title, F, (4,4), F, String, JSON, ε, ε, ε )
    }, ε, ε ),
  }, (_+), ε ),
}, ε, ε )
```

XML document Invoice

```
( invoice, U, (2,2), U, Map+Array, XML, {
  ( creationDate, F, (2,2), F, DateTime, XML, ε, ε, ε ),
  ( customerId, F, (1,1), F, Long, XML, ε, ε, ε ),
  ( dueDate, F, (1,1), F, DateTime, XML, ε, ε, ε ),
  ( @invoiceNo, T, (2,2), T, String, XML, ε, ε, ε ),
  ( items, U, (2,2), F, Array, XML, {
    ( product, F, (5,2), F, Map, XML, {
      ( @price, F, (5,5), F, String, XML, ε, ε, ε ),
      ( @productId, F, (5,5), F, String, XML, ε, ε, ε ),
      ( @quantity, F, (5,5), F, String, XML, ε, ε, ε ),
      ( @title, F, (5,5), F, String, XML, ε, ε, ε )
    }, ε, ε ),
  }, (product+), ε ),
  ( paid, F, (1,1), F, Boolean, XML, ε, ε, ε ),
  ( totalPrice, F, (2,2), F, Map+Array, XML, {
    ( __TEXT__, F, (2,2), F, Double, XML, ε, ε, ε )
    ( @currency, F, (2,2), F, String, XML, ε, ε, ε )
  }, (__TEXT__), ε ),
  ( vendorId, F, (1,1), F, Long, XML, ε, ε, ε ),
}, ((customerId|vendorId)+, creationDate,
dueDate?, items, totalPrice, paid?), ε )
```

Figure 4.11: An example of RSDs

- local ICs, various types of redundancy (both intra-model and inter-model), and intra/inter-model references,
- (partial) schemas required/allowed in selected models or inferred single-model schemas,
- possible, but not compulsory user interaction involving modification of suggested candidates (for ICs, redundancy etc.) and specification of non-detected cases, and
- processing of Big Data, i.e., maximum possible reduction of unnecessary information and parallel processing.

The input of the inference process is formed of the following:

1. A non-empty set of single/multi-model DBMSs $\mathcal{D}_1, \mathcal{D}_2, \ldots$ which together contain a set of kinds $\kappa_1, \ldots, \kappa_N$. Each kind is associated with its model(s). For each model supported in a particular DBMS $\mathcal{D}_i$ we also know whether

it is schema-less/full/mixed and whether the order of sibling properties of a kind must be preserved.

2. A (possibly empty) set of predefined schemas $\sigma'_1, \sigma'_2, ..., \sigma'_n$, $n \leq N$, (partially) describing selected kinds.

3. A (possibly empty) set of user-specified input information which can be of the following types:

   (a) A *redundancy set of kinds* $RK = \{\kappa_1, \kappa_2, ..., \kappa_r\}, r \leq N$ which describe the same part of reality, i.e., they will have a common schema σ. (Note that there is no restriction on the models the kinds in R can have. On the other hand, we do not know the schema of all kinds at this stage, so the redundancy cannot be specified at a higher level of granularity.)

   (b) A *simple data type* assigned to a selected property.

   (c) A *local IC* assigned to a selected property. The possible constraints involve identifier, unique, or (not) null.

   (d) A *reference* represented by an ordered pair of properties where the first one is the *referencing property* and the second one is the *referenced property*.

In other words, the user specifies the data on which the inference process should be applied. Depending on the type of the database system where it is stored, i.e., its specific features, the inference process can re-use an existing user-defined schema, it knows whether the order of siblings should be preserved, etc. Eventually, at the beginning or during the inference process (see Section 4.5), the user can provide a partial inferred schema, user-specified simple data types, ICs, and references for selected properties, as well as redundantly stored kinds.[13]

The general workflow of the inference process has two main phases – local and global. In the local phase, the process assumes as the input a large set of data, and the task is to reduce the information in parallel efficiently, i.e., we infer basic local schemas for each kind. The aim of the global phase is to merge the local schemas and enrich them with additional information gathered from the input data (i.e., ICs and references). Since we can assume that the number of all kinds in the whole multi-model schema is several orders smaller than the amount of input data, this phase does not need to be parallelised.

The workflow consists of the following stages:

1. *Local Schema Inferrer*: For each kind κ we generate its local RSD as follows:

   (a) If κ has a predefined schema σ'_κ, we transform it into RSD representation.

   (b) Otherwise, we generate for κ a *basic RSD* using a parallel approach as follows:
   (i) We generate a *trivial RSD* for each record (or property, depending on the selected type of the algorithm – see Sections 4.4.3 and 4.4.4) of κ.
   (ii) We merge (see Section 7 and 11) trivial RSDs of κ and eventually all kinds in its respective redundancy set RK_κ to a basic RSD.

[13] Note that for the sake of simplicity we currently consider redundancy only for whole kinds.

2. *Gathering of Footprints*: Parallel to the local schema inference, for each kind κ we gather its auxiliary statistics, called *footprints* (see Section 4.4.5), as follows:

   (a) *Phase Map*: We gather footprints for each value of each property p_i^κ of κ.

   (b) *Phase Reduce*: We merge all footprints of values of each property p_i^κ, resulting in an aggregated footprint of property p_i^κ.

   (c) *Candidate Set*: We apply a set of rules on the merged footprints of each property to produce a set of candidates for redundancy, local ICs, and references. Note that when the structure of kinds is inferred, the redundancy can be specified at the level of (complex) properties. A *redundancy set of properties* $RP = \{\pi_1, \pi_2, ..., \pi_s\}$, $s \in \mathbb{N}$, where each property is a part of some kind regardless its model, describes the same part of reality.

   (d) The user can confirm/refute the candidates at this stage or add new ones.

3. *Global Schema Inferrer*: Having a unified RSD representation and footprints for each input kind κ, we generate the final multi-model schema as follows:

   (a) (Optionally), we perform the full check of candidates. It is not done if the user confirms the candidates.

   (b) We merge all redundancy sets of properties, i.e., for each property $\pi_i \in RP_j$, $i, j \in \mathbb{N}$ we extend its schema by joining RSDs of all properties in RP_j.

   (c) We extend the RSDs with discovered references.

   (d) We create the final multi-model schema formed by all inferred RSDs.

4. We transform the resulting set of RSDs and respective ICs to the user-required output.

Next we introduce two versions of the local inference algorithm – *record-based* and *property-based*. The former follows the usual strategy in the existing works, i.e., "horizontal" processing; the latter introduces a "vertical" optimisation for complex data.

4.4.3 Record-Based Local Inference Algorithm

The more intuitive approach, so-called *Record-Based Algorithm* (RBA), considers a record, i.e., the root property including all its child properties, as a working unit. The input of Algorithm 4.1 consists of the particular database wrapper w_D (implementing specific behaviour and features of the system D) and set N_D of names of kinds whose schemas are to be inferred. Having initiated an empty schema S (i.e., a forest of RSDs), the algorithm infers the schema of each kind κ during three logically different steps:

- *Preparation phase*: The data is first loaded using a particular database wrapper w_D, and then each record is *in parallel* mapped into an RSD describing its trivial schema. The result of the preparation phase is a collection R (possibly containing duplicities) of RSDs describing schemas of individual records within kind κ.

- *Reduce phase*: Next, the collection R is merged using function *merge()* (see Section 7) *in parallel* into single r_κ describing the basic RSD of kind κ.

- *Collection phase*: The inferred schema r_κ is added to set S.

Having all the kinds processed, the resulting schema S is returned.

Algorithm 4.1: Record-Based Local Inference Algorithm

```
   Input: w_D – a wrapper for database system (or model) D
1          N_D – a set of names of kinds whose schema is to be inferred
   Output: S – a minimal set of structurally different RSDs
2  Schema S := ∅
3  foreach name_κ in N_D do
       // preparation phase:
4      R := w_D.mapRecordsToRSDs(name_κ)
       // reduce phase:
5      r_κ := R.merge()
       // schema collection phase:
6      S.add(r_κ)
7  return S
```

Merging of RSDs – Function merge()

During the merging process we merge the information, and we modify respectively the regular expression describing the data. In particular, having two RSDs $rsd_1 = (name_1, unique_1, share_1, id_1, types_1, models_1, children_1, regexp_1, ref_1)$ and $rsd_2 = (name_2, unique_2, share_2, id_2, types_2, models_2, children_2, regexp_2, ref_2)$, the merging process creates the resulting RSD $rsd = (name, unique, share, id, types, models, children, regexp, ref)$ as follows:

- Within local stage, names $name_1$, $name_2$ are always equal, i.e., only properties having the same name are merged, therefore $name := name_1$.

- *unique* is set to minimum value of $unique_1$, $unique_2$, where `F` $<$ `U` $<$ `T`. In other words, the fact that a property is not unique (i.e., either $unique_1$ or $unique_2$ is set to `F`) cannot be changed. If neither $unique_1$ nor $unique_2$ is set to `F` but at least one is set to `U`, we have to wait to finish parallel checking of the data. Otherwise, having $unique_1$ and $unique_2$ set to `T`, the resulting *unique* is set to `T`.

- *share* is sum of particular shares, i.e., $share := (share_p, share_x)$, where $share_p := share_{p_1} + share_{p_2}$ and $share_x := share_{x_1} + share_{x_2}$.

- Similarly to *unique*, the same principle applies for *id*.

- $types := types_1 \diamond types_2$ (see Section 4.4.1).
- $models := models_1 \cup models_2$.
- $children := children_1 \cup children_2$, whereas the child properties with the same name are recursively merged too.
- $regexp$ is the result of merging of regular expressions $regexp_1$ and $regexp_2$ using existing verified schema inference approaches [78, 156].[14] If $regexp_1 = \epsilon$, then $regexp := regexp_2$. Else $regexp := regexp_1$.
- If $ref_1 = ref_2$, then $ref := ref_1$. Otherwise, it has to be resolved by the user/default settings.

Algorithm 4.2: Function *merge()*

Input: r_1 , r_2 – RSDs to be merged
Output: r – the merged RSD

```
   // the names are always equal
1  r.name := r1.name
   // select minimum, i.e., F < U < T
2  r.unique := min(r1.unique, r2.unique)
   // sum shares
3  r.share := sum(r1.share, r2.share)
   // select minimum, i.e., F < U < T
4  r.id := min(r1.id,r2.id)
   // type merge operator
5  r.types := r1.types ◇ r2.types
   // union of models
6  r.models := r1.models ∪ r2.models
   // recursively merge children
7  r.children := mergeChildren(r1.children,r2.children)
   // regular expression merge
8  r.regexp := mergeRegexp(r1.regexp, r2.regexp)
   // references are the same or missing, therefore arbitrary ref is
      selected
9  r.ref := r1.ref
10 return r
```

4.4.4 Property-Based Local Inference Algorithm

Alternatively, instead of a whole record, the working unit of the local inference process can be a single property, whose eventual child properties are processed separately too. This strategy is not common in the existing approaches; however, it may lead to a performance boost (as we show in Section 4.6) when, e.g., the record is highly structured and contains a large number of nested properties. Moreover, this version of the algorithm can be merged with the mining of footprints (see Section 4.4.5) into a single algorithm.

[14]Due to rich related work (see Section 4.2) we omit technical details.

To work with individual properties instead of records, we need to be able to distinguish their RSDs. Therefore, we introduce the notion of a *hierarchical name* that uniquely identifies each property p by concatenating the names of properties on the path from the root property to property p, where each step is separated by a delimiter '/' (slash). As the 0th and 1st steps, we use the name of the system and the kind where the property occurs. Additionally, if the property is an anonymously named element of an array, the name of the property consists of '_' (underscore) and its data type.

Example 4.14. For example, property *productId* from the JSON document in Figure 4.9 has hierarchical name: `/mongoDB/Order/items/_Object/productId`
□

The input of the *Property-Based Algorithm (PBA)* (see Algorithm 4.3) also consists of the particular database wrapper w_D and set N_D of names of kinds whose schemas are to be inferred. Having initiated an empty schema S, the algorithm processes each kind κ as follows:

- *Preparation phase*: For each property p of each record, the hierarchical name of the property and the trivial RSD of the property is extracted from the data *in parallel*, forming a collection NP of pairs $(name_p, rsd_p)$. Next, the grouping according to $name_p$ is performed, resulting in the set GP of pairs $(name_p, P)$, where P is a set of RSDs of properties with the same $name_p$.

- *Reduce phase*: Next, each collection P is *in parallel* aggregated using function *aggregateByHierarchicalName()* (see Section 9) into rsd_p describing the basic RSD of property p with hierarchical name $name_p$. The resulting set of pairs $(name_p, rsd_p)$ is denoted as AP.

- *Collection phase*: Set AP is iterated and each rsd_p is added into schema S, continuously enriching and building the schema of kind κ using function *addToForest()* (see Section 11).

Finally, the resulting schema S is returned as the result.

Aggregating of RSDs – Function aggregateByHierarchicalName()

Generation of a basic (local) RSD consists of generating an RSD for each property and their aggregation into a common property schema. During the process, we aggregate the information, and we modify respectively the regular expression describing the order of the nested properties.

As we can see in Algorithm 4.4, having a collection of RSDs $rsd_i = (name_i$, $unique_i$, $share_i$, id_i, $types_i$, $models_i$, $children_i$, $regexp_i$, $ref_i)$, $i = 1, ..., n$, the aggregation process creates the resulting RSD $rsd_p = (name$, $unique$, $share$, id, $types$, $models$, $children$, $regexp$, $ref)$ corresponding to a schema of property p as follows:

- the names $name_i$ are always equal, i.e., only the properties having equal hierarchical name are aggregated, therefore $name := name_1$.

- *unique* is set to minimum value of $unique_i$, $i = 1, ..., n$, where $\texttt{F} < \texttt{U} < \texttt{T}$.

```
Algorithm 4.3: Property-Based Local Inference Algorithm
  Input: w_D – wrapper for database system (or model) D
1        N_D – set of names of kinds whose schema is to be inferred
  Output: S – a set of RSDs describing the resulting schemas of kinds in N_D
2 Schema S := ∅
3 foreach name_κ in N_D do
      // preparation phase:
4     NP := W_D.flatMapRecordsToPairs(name_κ)
      // group properties with the same name_p together
5     GP := NP.groupByKey()
      // reduce phase:
6     AP := GP.aggregateByHierarchicalName()
      // schema collection phase:
7     foreach (name_p, rsd_p) in AP do
8         addToForest(rsd_p, S)
9 return S
```

- *share* is set to the sum of shares, i.e., $share := (\sum_{i=1}^{n} share_{p_i}, \sum_{i=1}^{n} share_{x_i})$
- Similarly to *unique*, the same principle applies for *id*.
- $types := types_1 \cup ... \cup types_n$, whereas if there appear two types $t_i, t_j \in types$, s.t. $t_i \subset t_j$, then t_i is removed from *types*.
- $models := \bigcup_{i=1}^{n} models_i$.
- *regexp* is the result of merging regular expressions $regexp_1, \ldots, regexp_n$ using an existing verified approach.
- Within the aggregate function, *children* is always an empty set as individual children have their own hierarchical name and thus are processed separately. Its content is resolved in later stage of the algorithm within function *addToForest()* (see Section 11).
- For all $ref_1, \ldots, ref_n$ either $ref_i = \epsilon$ or all the values of ref_i are equal, therefore $ref := ref_1$ is selected. If $ref = \epsilon$, the references are resolved after applying the candidates stage (see Section 4.4.6).

Forest Appender – function addToForest()

The purpose of this function (see Algorithm 4.5) is to join RSDs describing the schema of particular properties to form an RSD corresponding to a schema of the whole kind. Moreover, the RSDs describing a schema of a single kind are grouped into a forest. Having pairs $(name_p, rsd_p)$ which are alphabetically ordered according to $name_p$ (in ascending order), the parent property is always included in the schema S before its children (note that locally the schema is a tree). If the properties are not ordered, then if any parent property is missing, we can insert an empty placeholder of the not-yet-processed parent allowing it to include its children. As soon as the parent is being processed, it replaces the placeholder.

Algorithm 4.4: Function *aggregateByHierarchicalName()*

Input: P – Non-empty list of Property RSDs to be aggregated
Output: rsd_p – resulting aggregated rsd

```
1  rsd_p := (null, T, (0,0), T, ∅, ∅, ∅, null, null)
   // the names are always equal
2  rsd_p.name := P.first().name
3  foreach property p in P do
       // select minimum, i.e., F < U < T
4      rsd_p.unique := min(rsd_p.unique, p.unique)
       // sum shares
5      rsd_p.share := sum(rsd_p.share, p.share)
       // select minimum, i.e., F < U < T
6      rsd_p.id := min(rsd_p.id,p.id)
       // type merge operator
7      rsd_p.types := rsd_p.types ◇ p.types
       // union of models
8      rsd_p.models := rsd_p.models ∪ p.models
       // regular expression merge
9      rsd_p.regexp := regexpMerge(rsd_p.regexp, p.regexp)
       // children are resolved later (using forest appender), the
          processed property contains children = ∅
   // references are the same or missing, therefore arbitrary ref is
      selected
10 rsd_p.ref := P.first().ref
11 return rsd_p
```

Algorithm 4.5: Function *addToForest()*

Input: $name_p$ – hierarchical name of a property p

```
1          o_p – object that holds an information, e.g., an RSD corresponding to
   property p
2          O – resulting set of possibly interlinked hierarchical objects, e.g., a
   forest of RSDs
3  name_κ := name_p.kind()
4  node := O.getOrCreateRoot(name_κ)
   // the first step of the hierarchical name
5  name := name_p.head()
6  repeat
7      if node.hasChildren(name) then
8          node := node.getChildren(name);
9      else
10         node := node.getOrCreateChildren(name)
11     if name is name_p.tail() then
           // if node represents an inner node, its content is merged
           // otherwise (leaf) its placed as is
12         node.placeContent(o_p)
       // the next step of the hierarchical name
13     name := name.next()
14 until name is name_p.tail()
```

4.4.5 Gathering Candidates for ICs and Redundancy

To detect integrity constraints and redundancy efficiently, we utilise a two-stage approach:

1. We efficiently detect a set of candidates.
2. The user can confirm/refute them or request a full check.

For this purpose, we introduce a set of lightweight and easy to compute *footprints* and we apply them to compute candidates for ICs (i.e., primary keys, intra- and inter-model references, and interval-based value constraints) and redundancy in data. A naive approach would compare active domains of all pairs of properties. Instead, when walking through all the data during the schema inference process, the same access can be exploited to mine statistical (and other) information about the active domains, i.e., the *footprints*. They can be then used to compare active domains and determine the desired integrity constraints more efficiently.

Property Domain Footprint (PDF)

For each property p, we define an active domain descriptor utilising basic statistics and the Bloom filter [108], so-called *Property Domain Footprint* (*PDF*). It is represented as a tuple $PDF = (count, first, unique, required, repeated, sequential, min, minHash, max, maxHash, totalHash, averageHash, bloomFilter)$.

- $count$ is the number of items of the active domain.
- $first$ is the number of parent properties in which p occurs.
- $unique \in \{\mathtt{T}, \mathtt{F}\}$ represents the uniqueness of values within a particular active domain. It is computed by counting the occurrence of each item of the active domain.
- $required \in \{\mathtt{T}, \mathtt{F}\}$ represents the nullability of the value of a particular property. It is computed by comparing $p.first$ of property p with $pp.count$ of its parent property pp, i.e., $required := (p.first = pp.count)$.
- $repeated \in \{\mathtt{T}, \mathtt{F}\}$ represents whether the property is a direct element of an array (`T`) or a single value (`F`). It is computed using auxiliary features $count$ and $first$, i.e., $repeated := (count \div first > 1)$.
- $sequential \in \{\mathtt{T}, \mathtt{F}, \mathtt{U}\}$ represents the possibility of an active domain of a simple property to represent a sequence of integers. The default value is `U` (if the property is not of data type `Integer`). The sequential feature is computed using auxiliary features min, max, and $count$, i.e., $sequential := (max - min = count - 1)$.
- min is the minimum value of the active domain.
- $minHash$ is the minimum hashed value of the active domain. It allows to compare values of distinct data types efficiently and prevents the comparison of possibly extensive data, e.g., BLOBs.

- max is the maximum value of the active domain.
- $maxHash$ is the maximum hashed value of the active domain.
- $totalHash$ is the sum of hashed values of the active domain.
- $averageHash := (totalHash \div count)$ represents the average of hash of unique values within the active domain.
- $bloomFilter$ is an array of "small" size σ describing a much larger active domain K of property p at the cost of false positives (i.e., equal hashes of two different values). Using multiple hash functions $H_1(), \ldots, H_n()$ returning values from $\{1, \ldots, \sigma\}$, each distinct value $k \in K$ is hashed and each value of $bloomFilter[H_i(k)]$ is incremented.

PDF Miner Algorithm

The purpose of the footprint miner (see Algorithm 4.6) is to create a PDF for each property. First, the data are loaded from the database store in the form of records using a particular database wrapper. For each property p of each individual record a footprint f is created describing a single value of active domain of a certain property. The hierarchical name $name_p$ is attached to each footprint instance. Next, the instances are merged to create distinct unique sets of each active domain using function *mergeValueDuplicates()*. Then, the distinct values are first grouped by function *groupByKey()*, resulting in set GP of pairs $(name_{p_i}, F_i)$, where $F_i = \{f_{i_0}, \ldots, f_{i_n}\}$ is the set of footprints. Second, they are grouped to determine the footprint f_p describing the whole active domain. To do so, merge function *aggregateByHierarchicalName()* (see Algorithm 4.7) is applied.

Algorithm 4.6: PDF Miner Algorithm

```
   Input: W_D – wrapper for database system (or model or data source in
          general) D
 1        N_D – set of names of kinds whose schema is to be inferred
   Output: S – a set of hierarchically ordered footprints
 2 SchemaForest S = ∅
 3 foreach name_κ in N_D do
       // preparation phase:
 4     NP := W_D.flatMapRecordToNameFootprintPairs(name_κ)
       // aggregate footprints to create one footprint for each
          property:
 5     DP := NP.mergeValueDuplicates()
 6     GP := DP.groupByKey()
 7     AP := GP.aggregateByHierarchicalName()
       // footprint finalisation and collection phase:
 8     foreach (name_p, pdf_p) in AP do
 9         addToForestAndFinalize(pdf_p, S)

10 return S
```

Finally, the aggregated property features are appended to the tree structure representing the data and the missing features (i.e., *repeated*, *sequential*, and $averageHash$) are resolved.

Algorithm 4.7: Function *aggregateByHierarchicalName()*

```
   Input: F – Non-empty list of footprints to be aggregated
1  r := (0, 0, T, T, F, F, ∞, ∞, 0, 0, 0, 0, [0, . . . , 0])
2  foreach footprint f in F do
3      r.count := sum(r.count, f.count)
4      r.first := sum(r.first, f.first)
5      r.unique := r.unique AND f.unique
6      r.required := r.required AND f.required
       // repeated and sequential are resolved in later stages
7      r.min := min(r.min, f.min)
8      r.minHash := min(r.minHash, f.minHash)
9      r.max := max(r.max, f.max)
10     r.maxHash := max(r.maxHash, f.maxHash)
11     r.totalHash := sum(r.totalHash, f.totalHash)
       // averageHash is resolved in later stages
12     r.bloomFilter := merge(r.bloomFilter, f.bloomFilter)
13 return r
```

Candidate Builder Algorithm

Having computed footprint f_p for each property p, we can determine candidates for identifiers, references, and data redundancy. We propose Algorithm 4.8 that consists of three phases:

1. *Identifier Candidate:* The candidates for identifiers C_{ident} are inferred from the footprints in set F. An identifier must be unique within the active domain of the respective property p and required. Also, the property can not be a direct element of an array. Therefore, the algorithm tests whether $f_p.unique = f_p.required = \texttt{T}$ and $f_p.repeated = \texttt{F}$.

2. *Reference Candidate:* Candidates for references C_{ref} are inferred on the basis of several observations. A reference is a property that refers to the identifier of another kind. Therefore, we search the Cartesian square $F \times C_{ident}$ excluding pairs (c, c), $c \in C_{ident}$ in order to find pairs (f, c), s.t. f is the footprint of the referencing property p_f and c is the footprint of the referenced property p_c. Additionally, the active domain of referencing property p_f must form a subset of active domain of the referenced property p_c. Therefore, we compare active domains of both the properties using function *formsSubset()* (see Algorithm 4.9), i.e., we analyse footprints f and c using the following rules:

    - The referencing property does not have to be strictly unique, as the one-to-many (as well as many-to-one or many-to-many) relationship may occur.
    - The referencing property does not have to be required as the lower bound of relationship may be zero-to-one/many.
    - It must hold that $f.minHash \geq c.minHash$, i.e., the referencing property p_f does not contain a smaller value than the referenced property p_c.

Algorithm 4.8: Candidate Builder Algorithm

```
   Input: F – list of (pre)computed footprints
 1        k – minimum number of siblings/descendants to form a redundancy
 2 C_ident := ∅; C_ref := ∅; C_red := ∅;
   // Identifier candidate phase:
 3 foreach footprint f in F do
 4     if f.unique AND f.required AND not f.repeated then
 5         C_ident.add(f)
   // Reference candidate phase:
 6 foreach identCandidate c in C_ident do
 7     foreach footprint f in F \ c do
 8         r := formsSubset(f, c)
 9         if r is not "EMPTY" then
10             if isAutoincrement(f, c) then
11                 C_ref.add((f,c,"WEAK",r))
12             else
13                 C_ref.add((f,c,"STRONG",r))
   // Redundancy candidate phase:
14 foreach refCandidate (f, c, t, r) in C_ref do
15     D_f := descendantOrSibling(f);
16     D_c := descendantOrSibling(c);
17     R := ∅
18     red := "FULL"
19     foreach d_1 in D_f do
20         foreach d_2 in D_c do
21             type := formsSubset(d_1, d_2)
22             if type is not "EMPTY" then
23                 R.add(d_1,d_2)
24                 red := min(red, type)
25     if R.size() ≥ k then
26         R.add((f,c))
27         C_red.add((R,min(red, r)))
28         t := "WEAK"
29 return (C_ident, C_ref, C_red)
```

- Similarly, it must hold that $f.maxHash \leq c.maxHash$.
- Finally, only if all the above conditions are satisfied, Bloom filters are compared. To denote that property f is a reference to property c for each pair of elements $f.bloomFilter[i]$, $c.bloomFilter[i]$ it must hold that $f.bloomFilter[i] \leq c.bloomFilter[i]$. In other words, active domain of p_f must be a subset of active domain of p_c.

Additionally, we distinguish between *strong* and *weak* reference candidates. Having *sequential*, *unique*, and *required* set to `T` for both the referencing and referenced properties may imply that there is no relationship between

the properties (i.e., both properties form a technical (auto-incremented) identifier of their kind). Therefore, such a combination may lead to a weak candidate for a reference.[15]

3. *Redundancy Candidate:* Finally, reference candidates may be extended into data redundancy candidates C_{red}. Naturally, each pair of referencing and referenced properties having footprints f and c store redundant information. However, we assume that redundantly stored data should cover a more significant part. Hence, we check their descendants and siblings to find more pairs of properties whose active domains form a mutual subset. If there is at least k pairs of neighbouring properties forming such subsets, the reference candidate (f, c) is marked as a *weak reference candidate* and, together with its neighbourhood, extended into a *redundancy candidate*. If for all the pairs of properties in the redundancy candidate the active domains are equal, we speak about *full redundancy*. Otherwise, i.e., when one kind contains only a subset of records of another kind, it is a *partial redundancy*. Also, note that only redundant properties are considered as a part of redundancy, even though both kinds may contain properties having the same name. If multiple properties can form a redundancy pair with the same property, it is up to the user to decide.

Algorithm 4.9: Function *formsSubset()*

Input: f_1, f_2 – compared footprints
// $f_1.minHash = f_2.minHash \rightarrow$ `"FULL"`
// $f_1.minHash > f_2.minHash \rightarrow$ `"PARTIAL"`
// $f_1.minHash < f_2.minHash \rightarrow$ `"EMPTY"`
1 $minType$:= determine($f_1.minHash$, $f_2.minHash$)
// $f_1.maxHash = f_2.maxHash \rightarrow$ `"FULL"`
// $f_1.maxHash < f_2.maxHash \rightarrow$ `"PARTIAL"`
// $f_1.maxHash > f_2.maxHash \rightarrow$ `"EMPTY"`
2 $maxType$:= determine($f_1.maxHash$, $f_2.maxHash$)
// $f_1.averageHash = f_2.averageHash \rightarrow$ `"FULL"`
// $f_1.averageHash <> f_2.averageHash \rightarrow$ `"EMPTY"`
3 $avgType$:= determine($f_1.averageHash$, $f_2.averageHash$)
// $\forall i : f_1.bloomFilter[i] = f_2.bloomFilter[i] \rightarrow$ `"FULL"`
// $\forall i : f_1.bloomFilter[i] <= f_2.bloomFilter[i] \rightarrow$ `"PARTIAL"`
// $\exists i : f_1.bloomFilter[i] > f_2.bloomFilter[i] \rightarrow$ `"EMPTY"`
4 $bfType$:= determine($f_1.bloomFilter$, $f_2.bloomFilter$)
5
6 **if** min($minType$,$maxType$,$avgType$,$bfType$) is "FULL" **then**
7   **return** "FULL"
8 **else if** min($minType$,$maxType$,$bfType$) is "EMPTY" **then**
9   **return** "EMPTY"
10 **else**
11   **return** "PARTIAL"

[15]Note that such a candidate is not discarded, yet it is not marked as recommended when the inference process applies candidates for RSDs inferred within the local stage.

Example 4.15. Figure 4.12 introduces an example of footprints of selected properties of the multi-model data from Figure 4.9 and their application for detection of candidates. The upper part of the figure contains the data, i.e., a subset of properties from relational table *Product* (violet), nested documents _ from JSON collection *Order* (green) and nested elements *Product* from XML collection *Invoice* (grey). The bottom three parts correspond to the three phases, where we can see the values of respective necessary features of footprints.

In order to determine all the footprint features, the duplicate values are removed, the unique values are hashed (using, e.g., rolling hash $rh(v) := v_{rh}$ for each $v \in \mathbb{V}$) and then *minHash*, *maxHash*, *averageHash*, and *bloomfilter* are computed as was described. *averageHash* is rounded to $floor(averageHash)$ and to compute the Bloom filter (of size = 2), hash functions $h_1(v_{rh}) := v_{rh} \mod 2$ and $h_2(v_{rh}) := floor(sqrt(v_{rh})) \mod 2$ are used.

Regarding the detection of candidates, first all footprints are iterated and their *unique*, *required*, and *repeated* features are checked. In this particular case, only the footprint of property *id* satisfies that *unique* and *required* are both set to `T` and *repeated* is set to `F`, therefore only a single identifier candidate is created and propagated to the next phase of the algorithm. Also note that *id* is probably a technical identifier based on auto-increment (i.e., *sequential* is also set to `T`).

Next, all relevant distinct pairs of footprints are compared to find candidates for references. If any footprint feature does not satisfy the requirements for a reference, it is denoted by the red colour. As we can see, only properties *productId* (JSON) and *productId* (XML) satisfy the requirements and therefore form the set of reference candidates C_{ref}.

Finally, reference candidates are checked to form redundancy candidates, whereas $k = 2$. In this case, we compare siblings (as there are no further nested properties) of pairs (*id*, *productId*) (JSON) and (*id*, *productId*) (XML). In the former case, there is a redundancy between properties *title* (REL, JSON), *brand* (REL, JSON), and *price* (REL, JSON). In the latter case, there is a redundancy only between properties *title* (REL, XML). In the former case, the number of pairs $3 \geq k$, therefore, the reference candidate is extended into the redundancy candidate, and the former reference candidate is marked as weak. In the latter case, the reference candidate remains unchanged as $1 < k$, and it does not form the candidate for redundancy.

Also note that $(id, title, price, brand)$ (REL), $(productId, title, price, brand)$ (JSON) form a partial redundancy, since multiple requirements are violated, e.g., features *average* are not equal (see the bold font). □

4.4.6 Global Phase

The local phase consists of inference of local (single-system, single-kind) schemas described as tree-based RSDs and building a set of candidates for identifiers, references, and redundancy. The global phase applies the knowledge gained in the previous steps and joins RSDs using candidates for references and redundancy into the resulting global multi-model schema. It can also begin with an optional full check of candidates, i.e., removing false positives.

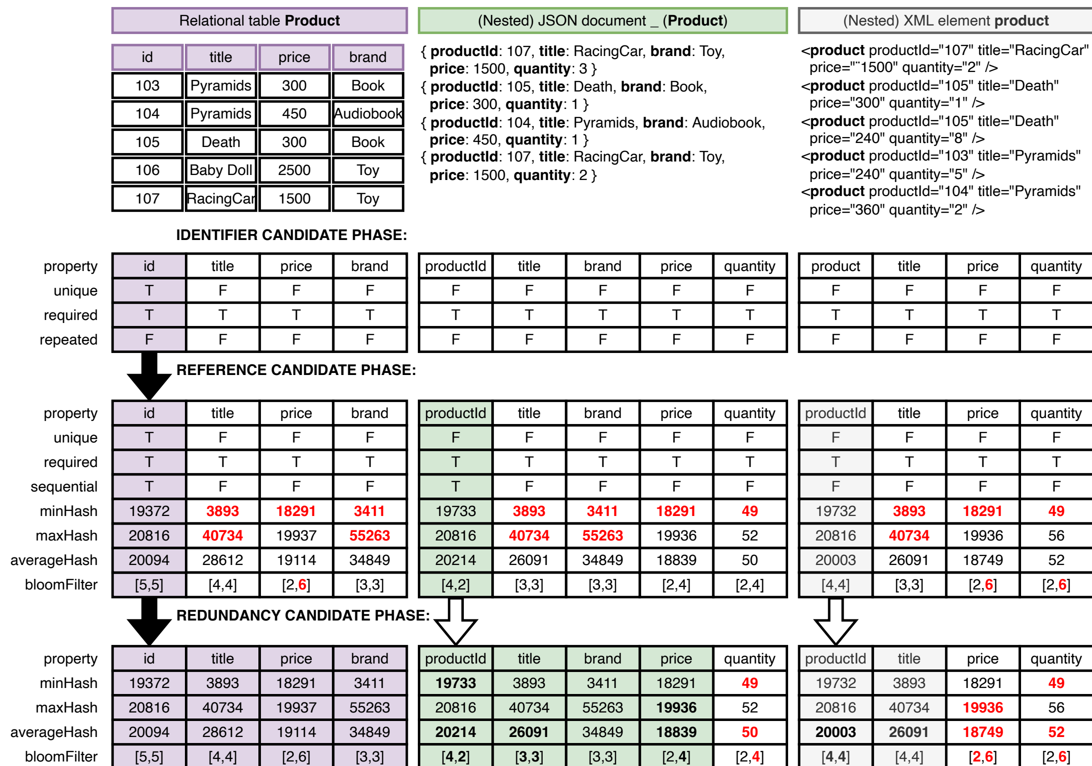


Figure 4.12: An example of selected footprints and building of candidates

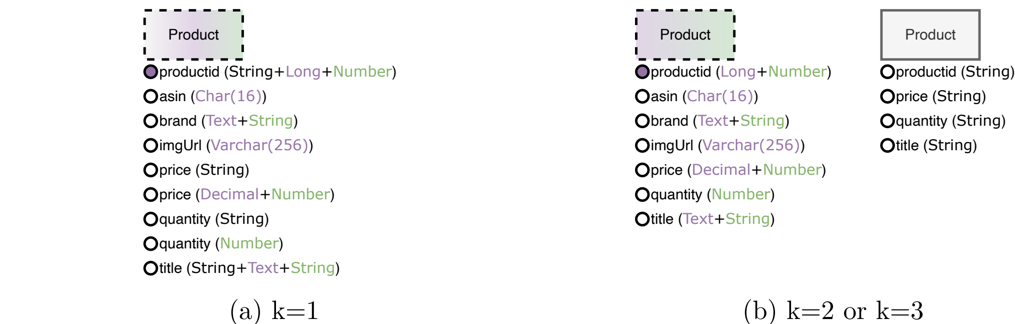


Figure 4.13: Example of redundancy, k=1, k=2, k=3

Checking of Candidates

Depending on the implementation, either the user may confirm/refute the suggested candidates (or denote user-specified candidates) using an interactive interface (see Section 4.5). Or, (s)he may decide which subset of candidates for references and redundancy[16] will be thoroughly checked. The checking itself is implemented as a distributed MapReduce job:

- Checking of references involves mapping of each value of a referencing property ref into tuple $(value, \texttt{REF})$ while each value of referenced property id is mapped to tuple $(value, \texttt{ID})$ as well. Reduction and further mapping by key takes place as follows:
  - If the list assigned to $value$ contains both `REF` and `ID`, the result is

[16]Candidates for identifiers do not have to be checked once approved by the user as long as the identifier requires only features *unique* and *requires* being set to `T`.

mapped to 0.

  - If the list assigned to $value$ contains only `ID`, the result is mapped to 1.
  - Otherwise, the result is mapped to -1.

  Finally, the minimum value is selected. If the result is -1, the candidate for reference is not valid and removed. If the result is 0, it is a full reference (i.e., the active domains of ref and id are equal). The result of 1 denotes that property ref refers to a subset of the active domain of property id.

- Checking of redundancy is similar; in addition we have to check values of all neighbouring redundant properties of the referencing and referenced properties ref and id. First, for each record its redundant properties are mapped to tuple $(value, \{red_subrecord, source\})$, where $value$ is the value of referencing/referenced property, $red_subrecord$ are values of ordered redundant properties in the record, and $source \in \{\texttt{REF}, \texttt{ID}\}$. Next, the tuples are reduced by key and then mapped as follows:

  - If the list assigned to $value$ contains two equal sub-records from distinct sources, the result is mapped to 0. If the sub-records are not equal, the result is mapped to -1.
  - If the list assigned to $value$ contains only sub-record from the $source$ `REF`, the result is mapped to -1.
  - Otherwise, the result is mapped to 1.

  Finally, the minimum value is selected. If the result is -1, the candidate for redundancy is not valid, i.e., either there is a sub-record in the kind containing the referencing property ref that is not a part of the kind containing referenced property id, or the sub-records with the same $value$ do not share the same values in all redundant neighbouring properties. If the result is 0, the redundancy is full. Otherwise, the redundancy is partial.

Joining of RSDs

Joining of RSDs may be implemented variously, depending on the selected output of the inference process. Either the RSDs, together with the confirmed/soundly checked candidates for identifiers, references, and redundancy, are transformed to an output format, such as XML Schema [104, 105], JSON Schema [106] etc. Or, a more abstract representation using ER [31] or UML [32] can be used. How the information is captured depends on the selected format. For example, for representation of redundancy, the globally defined XML elements and respective XML references can be used. In the implementation of the inference approach (see Section 4.5), we also use a simple visualisation of the forest of RSDs, where the identifiers, as well as redundant parts of the trees, are respectively graphically denoted, and the trees are interlinked with a new type of edges representing the references.

Example 4.16. Depending on the selected parameters of the algorithm, we can get as a result, e.g., the ER model depicted in Figure 4.1. If we look closely,

e.g., at entity *Product*, in Figure 4.13 we can see an example of its alternative result depending on parameter $k = 1$ or $k \in \{2, 3\}$. The colours represent the respective "overlapping" models and properties (relational, JSON document, or XML document). If $k = 1$, we would get one common schema for kind *Product* represented in all the three models. If $k \in \{2, 3\}$, we would get a common schema for the relational and JSON document model and a different schema for the XML document model.

Note that property *productId* is an identifier only in the relational table *Product* (illustrated by the purple colour). Also, note that non-redundant property *price* has probably a different meaning in distinct models. In the case of the XML document model, it could be a purchase price, whereas, in the relational and JSON document model, it could be a selling price. □

4.5 Architecture and Implementation

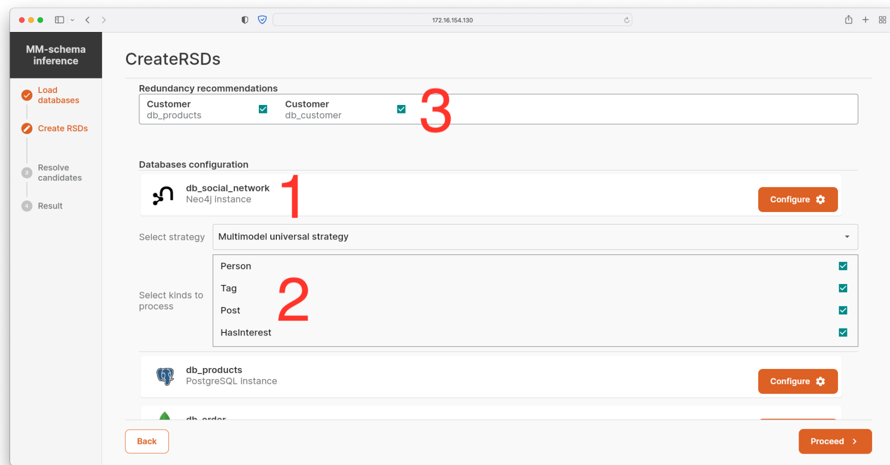
The proposed approach was implemented as a modular framework called *MM-infer*.[17] Its graphical interface and general functionality have been introduced in demo paper [109], but without technical details, algorithms, and experiments provided in this paper. It currently supports the following models and DBMSs: *PostgreSQL* (relational and document, i.e., multi-model), *Neo4j*[18] (graph), and *MongoDB*[19] (document) which represents both schema-full and schema-less DBMS.

The frontend of *MM-infer* was implemented in *Dart* using framework *Flutter*.[20] Sample screenshots are provided in Figure 4.14. The expected work with the tool is as follows: The user selects particular DBMSs (1) and kinds (2) to be involved in the inference process. (S)he can also confirm/refute initial redundancy (3) based on kind names. Then the local schema inferrer infers the local RSDs and generates the candidates. In the next screen, the user selects particular types of candidates (4) and confirms/refutes the suggestions (5). After the global schema inferrer performs the full check of candidates (if required) and merges the RSDs into the global schema to be visualised to the user or transformed to a requested form (6).

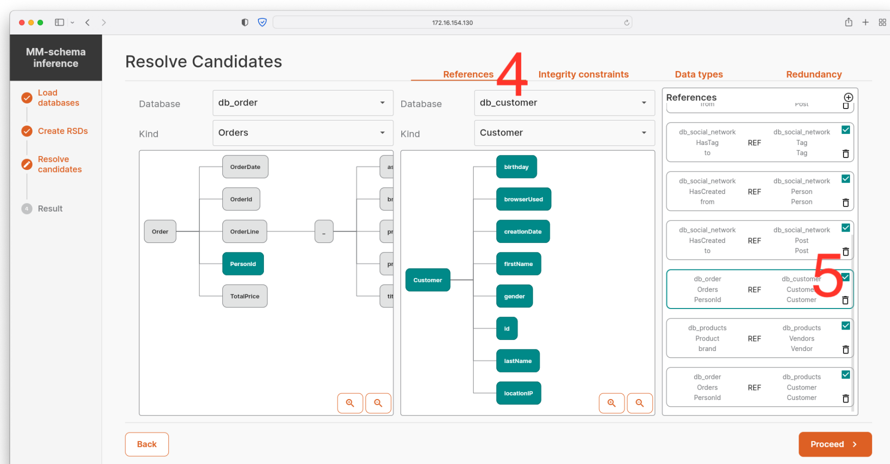
The backend of *MM-infer* was implemented in *Java* and using *Apache Spark*.[21] The architecture of *MM-infer*, depicted in Figure 4.15, reflects the steps of the above described inference process:

- At the bottom we can see data sources (green box) – a multi-model DBMS or a set of single/multi-model DBMS (i.e., a polystore-like storage).
- The local schema inferrer (yellow box) uses three types of wrappers that transform the input data/schemas into RSDs:
  - In the case of schema-full database, a *pre-defined schema* already exists. Therefore, we only fetch the schema and translate it to the unifying representation (i.e., only a unifying wrapper translator must be

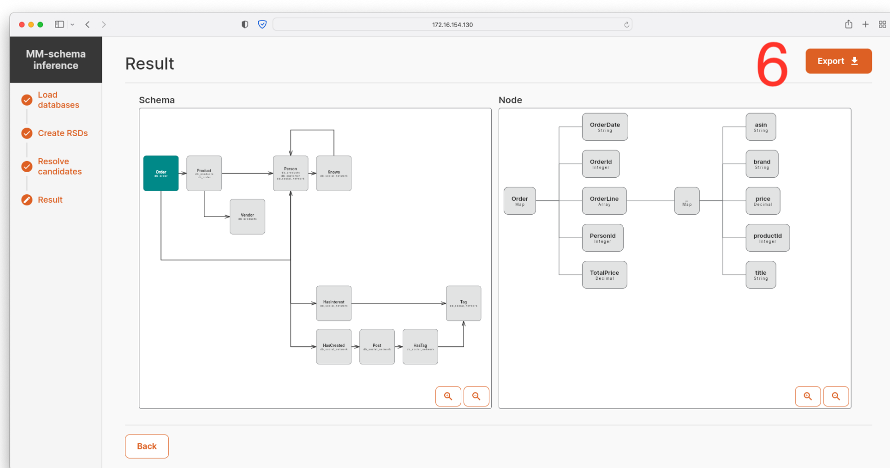
[17] https://www.ksi.mff.cuni.cz/~koupil/mm-infer/index.html
[18] https://neo4j.com/
[19] https://www.mongodb.com/
[20] https://flutter.dev/
[21] https://spark.apache.org/



(a)



(b)



(c)

Figure 4.14: Screenshots of *MM-infer*

implemented to translate the local schema into a unified representation).

– Having a schema-free (or mixed) database approach, a robust schema inference approach may already exist for a particular model (e.g., the XML or JSON document model). If so, we infer the schema by exploiting such an approach, and then we translate the resulting *inferred schema* using a unifying translator layer.

– Finally, having schema-free or schema-mixed DBMS and no existing schema inference approach, the *basic schema* is inferred for each kind.

Local schema inferrer then merges the RSDs locally (i.e., within one DBMS) using *Apache Spark*. In parallel, it gathers and merges the data statistics and produces the respective candidates to be eventually modified by the user.

- The global schema inferrer (red box) checks candidates for references and redundancy and merges the RSDs globally (i.e., in the context of all inputs).

- The resulting multi-model schema is provided to the user in the chosen representation (violet box).

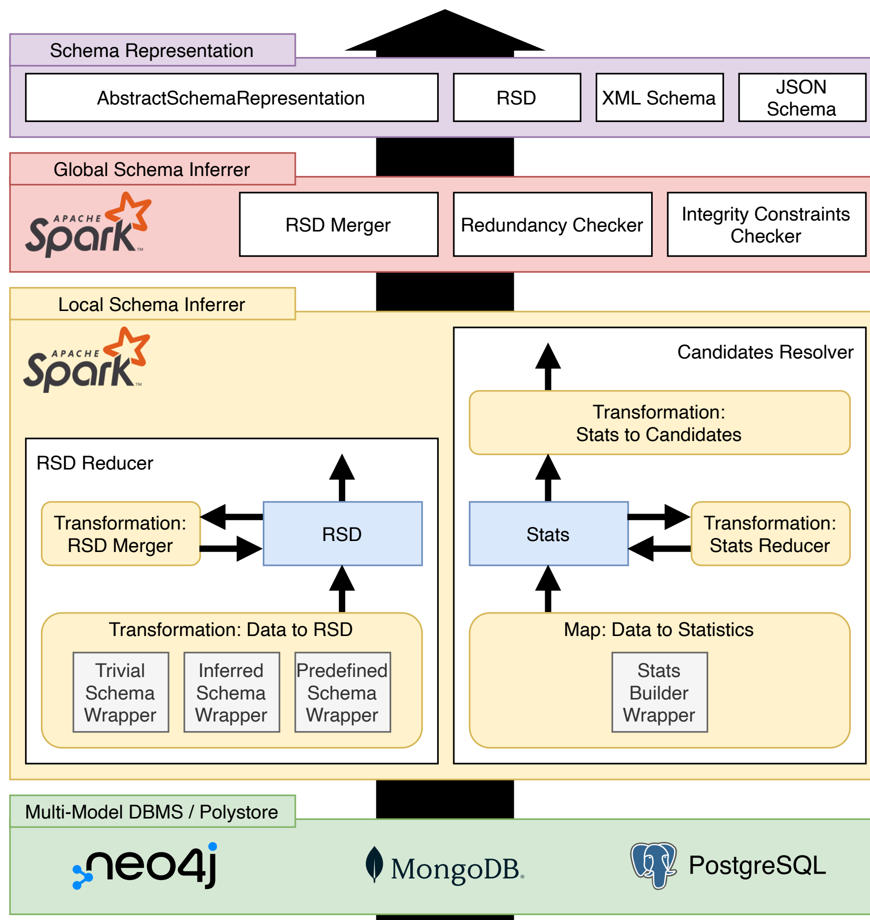


Figure 4.15: Architecture of *MM-infer*

4.5.1 Database Wrappers

MM-infer is based on custom-tailored wrappers, each of which reads individual records from a particular DBMS and returns its RSD. Note that not only the whole data set may be represented in different data formats and data models, but a collection of data models may represent even a single record. The wrapper also includes user settings that determine the level of detail of the described schema. For example, the user may request the inference of data structures which implicitly may not be supported by the data model (e.g., `Set` and `Map` in the case of JSON documents) but can be specified in RSDs.

Naturally, we assume the implementation of a wrapper for each DBMS. However, separate wrappers may exist for distinct settings of a particular DBMS – e.g., schema-less vs schema-mixed – or a wrapper that involves a particular single-model schema inference approach. A separate wrapper module is also devoted to reading data statistics and their transformation into PDFs.

The following examples show the core of sample implementation of wrappers for particular considered models of selected popular DBMSs.

Example 4.17. Table 4.2 illustrates the mapping of PostgreSQL record schema (representing relational tuple) or property (i.e., an attribute of a tuple or an identifier of a tuple) to an RSD. Note that PostgreSQL record (tuple) has an anonymous name _ and its type corresponds to `Map` as the properties are named values stored in an arbitrary order. In this case, we distinguish between `Simple` (representing any simple type) and `Array` type attribute, both possibly nullable (i.e., *share* can be either 0 or 1), and array being only homogeneous (i.e., described by a set of children $\{\dots\}$ and a trivial automaton representing an arbitrary number of anonymously named elements of an array (i.e., _+). Also, note that a structured property is not allowed as the purely relational model is naturally aggregate-ignorant.[22] Finally, we may consider an identifier and a reference as a particular attribute type. The identifier must be unique, and *share* cannot be 0 as the identifier cannot be nullable. In the case of references, the ref field is set to the referenced kind κ and respective referenced property p. □

Table 4.2: An example of *PostgreSQL* (relational model) schema mapping

construct	name	unique	share	id	types	models	children	regexp	ref
tuple	_	U	1,1	F	Map	REL	{...}	ϵ	ϵ
attribute (simple)	name	T/F/U	0/1,0/1	T/F/U	Simple	REL	ϵ	ϵ	$\epsilon/\kappa.p$
attribute (array)	name	T/F/U	0/1,0/1	F	Array	REL	{...}	trivial	ϵ
element of an array	_	T/F/U	1,0/1	F	Simple	REL	ϵ	ϵ	ϵ
identifier (simple)	name	T	1,1	T	Simple	REL	ϵ	ϵ	$\epsilon/\kappa.p$
reference	name	T/F/U	0/1,1	T/F/U	Simple	REL	ϵ	ϵ	$\kappa.p$

[22]PostgreSQL also supports the document model (i.e., JSON and XML). In this case, the mapping of respective embedded properties is the same way as in the document model (see Example 4.22 and 4.21).

Example 4.18. Table 4.3 illustrates mapping of SciDB[23] array schema to the unified RSD representation. A multi-dimensional array consists of anonymously named cells which contain a `Map` of named simple attributes. Note that SciDB does not allow complex attributes, therefore neither `Array` type nor `Structure` type is allowed in the mapping. Finally, each cell is uniquely identified by dimension coordination (i.e., an identifier), whereas references between tables are not supported. □

Table 4.3: An example of *SciDB* (array model) schema mapping

construct	name	unique	share	id	types	models	children	regexp	ref
cell	_	U	1,1	F	Map	ARRAY	{...}	ϵ	ϵ
attribute (simple)	name	T/F/U	0/1,0/1	T/F/U	Simple	ARRAY	ϵ	ϵ	ϵ
identifier (simple)	name	T	1,1	T	Simple	ARRAY	ϵ	ϵ	ϵ

Example 4.19. Table 4.4 illustrates mapping of Neo4j node/edge schema to RSD. Both a node and an edge have an anonymous name _ and its type corresponds to `Map` (i.e., both contain an unordered map of uniquely named properties). Similarly to PostgreSQL (its relational model), only simple attributes and homogeneous arrays of simple types are allowed. Also, only simple identifiers are allowed, and only two special kinds of references are allowed, i.e., special properties *from* and *to* representing the source and the target of an edge. Also, note that $share = 0$ is not allowed since *null* meta values are represented as a missing property in Neo4j. Therefore no property can be mapped to an RSD having $share = 0$. □

Table 4.4: An example of *Neo4j* (graph model) schema mapping

construct	name	unique	share	id	types	models	children	regexp	ref
node/ edge	_	U	1,1	F	Map	GRAPH	{...}	ϵ	ϵ
property (simple)	name	T/F/U	1,1	T/F/U	Simple	GRAPH	ϵ	ϵ	$\epsilon/\kappa.p$
property (array)	name	T/F/U	1,1	F	Array	GRAPH	{...}	trivial	ϵ
element of an array	_	T/F/U	1,0/1	F	Simple	GRAPH	ϵ	ϵ	ϵ
identifier (simple)	name	T	1,1	T	Simple	GRAPH	ϵ	ϵ	ϵ
reference	from/ to	T/F/U	1,1	F	REF	GRAPH	ϵ	ϵ	$\kappa.p$

Example 4.20. Table 4.5 illustrates mapping of Redis[24] (key/value model) key/value pair schema to RSD. A pair is an anonymously named `Tuple` of ordered anonymous properties, i.e., a key and a value. A key is a simple identifier while the value can be `Simple`, `Array`, `Set` or structured in general. However, we

[23] https://www.paradigm4.com/
[24] https://redis.io/

do not support mapping of structural values – otherwise it would be document model. Redis also does not support references, therefore no mapping exists for them.[25] □

Table 4.5: An example of *Redis* (key/value model) schema mapping

construct	**name**	**unique**	**share**	**id**	**types**	**models**	**children**	**regexp**	**ref**
pair	_	U	1,1	F	Tuple	KV	{...}	ϵ	ϵ
value (simple)	_	T/F/U	1,1	F	Simple	KV	ϵ	ϵ	ϵ
value (array)	_	T/F/U	1,1	F	Array	KV	{...}	auto-maton	ϵ
value (set)	_	T/F/U	1,1	F	Set	KV	{...}	ϵ	ϵ
element of an array/ set	_	T/F/U	1,0/1	F	Any-Type	KV	{...}/ϵ	ϵ	ϵ
key (simple)	_	T	1,1	T	Simple	KV	ϵ	ϵ	ϵ

Example 4.21. Table 4.6 illustrates the mapping of an XML schema to a unifying RSD. The root of an XML document is a named root element possibly having attributes (reflected as type `Map`) and nested subelements (reflected as type `Array`), or both (`Array+Map`). Any element can be `Simple` (i.e., without nested elements) or `Array`-type (i.e., having at least 1 nested element), with or without attributes. In addition, XML allows text content of an element (being arbitrary nested between other subelements within an array). As for the attributes, only simple types are allowed, and the name of an attribute is prefixed by `@`. Finally, a simple/composite attribute may identify an XML element. A reference may refer to this identifier. Both are mapped as an identifier or a reference as a special kind of attribute. Note that in an XML document, an element of an array is an ordinary named (sub)element. Therefore Table 4.6 does not contain a special row for an element of an array. □

Example 4.22. Table 4.7 illustrates the mapping of a JSON schema into a unifying RSD. The root of a JSON document is an anonymously named map (reflected as type `Map`) of name/value pairs (fields). A field can be simple (i.e., allowing only simple types of values), array (i.e., a homogeneous array allowing elements of the same type or a heterogeneous array allowing elements of any type), or structural (i.e., a nested document of type `Map`). An element of an array may be of any type, e.g., simple, array, or structural. Finally, each document is identified by an identifier (simple or composite), and references to other documents are supported. □

Example 4.23. Finally, Table 4.8 illustrates mapping of *Cassandra*[26] column family schema to a unifying RSD. A row of column family is an unordered set of uniquely named name/value pairs, i.e., reflected as type `Map`. In addition, *Cassandra* explicitly allows many variations of columns – e.g., a simple column,

[25] Yet it is not a rule in general, e.g., key/value store *RiakKV* (https://riak.com/products/riak-kv/index.html) supports links between so-called *buckets* allowing link walking

[26] https://cassandra.apache.org/

Table 4.6: An example of *MarkLogic* (XML document model) schema mapping

construct	name	unique	share	id	types	models	children	regexp	ref
root	name	U	1,1	F	Array/ Map/ Array + Map	XML	{...}	automaton	ϵ
attribute	@name	T/F/U	1,1	T/F/U	ϵ/ Simple	XML	ϵ	ϵ	$\epsilon/\kappa.p$
element (simple w/o atts)	name	T/F/U	1,0/1	F	Simple	XML	ϵ	ϵ	ϵ
element (simple + atts)	name	T/F/U	1,0/1	F	Array + Map	XML	{...}	(__TEXT__?)	ϵ
element (array w/o atts)	name	T/F/U	1,0/1	F	Array	XML	{...}	automaton	ϵ
element (array + atts)	name	T/F/U	1,0/1	F	Array + Map	XML	{...}	automaton	ϵ
text node	__TEXT__	T/F/U	1,0/1	F	ϵ/ Simple	XML	ϵ	ϵ	ϵ
identifier (simple)	name	T	1,1	T	Simple	XML	ϵ	ϵ	ϵ
reference	name	T/F/U	0/1,1	T/F/U	Simple	XML	ϵ	ϵ	$\kappa.p$

Table 4.7: An example of *MongoDB* (JSON document) schema mapping

construct	name	unique	share	id	types	models	children	regexp	ref
document	_	F	1,1	F	Map	DOC	{...}	ϵ	ϵ
field (simple)	name	T/F/U	0/1,1	T/F/U	ϵ/ Simple	DOC	ϵ	ϵ	$\epsilon/\kappa.p$
field (hom. array)	name	T/F/U	0/1,0/1	T/F/U	ϵ/ Array	DOC	{...}	trivial	ϵ
field (het. array)	name	T/F/U	0/1,0/1	T/F/U	ϵ/ Array	DOC	{...}	auto-maton	ϵ
field (structure)	name	T/F/U	0/1,0/1	T/F/U	ϵ/Map	DOC	{...}	ϵ	$\epsilon/\kappa.p$
element of an array	_	T/F/U	1,0/1	T/F/U	Any-Type	DOC	ϵ/{...}	ϵ/ trivial/ auto-maton	ϵ
identifier (simple)	name	T	1,1	T	Simple /Map	DOC	ϵ/{...}	ϵ	$\epsilon/\kappa.p$
reference	name	T/F/U	0/1,0/1	T/F/U	Simple /Map	DOC	ϵ/{...}	ϵ	$\kappa.p$

complex columns (e.g., array, set, map), or simple and complex identifiers. On the other hand, *Cassandra* does not allow references. Therefore no mapping for references is proposed (allowed). □

Table 4.8: An example of *Cassandra* (columnar model) schema mapping

construct	name	unique	share	id	types	models	children	regexp	ref
row	_	F	1,1	F	Map	COL	{...}	ϵ	ϵ
column (simple)	name	T/F/U	0/1,0/1	T/F/U	Simple	COL	ϵ	ϵ	ϵ
column (array)	name	T/F/U	1,1	F/U	Array	COL	{...}	auto-maton	ϵ
tuple	name	T/F/U	1,1	F/U	Tuple	COL	{...}	auto-maton	ϵ
column family	name	T/F/U	1,1	F/U	Map	COL	{...}	trivial	ϵ
column (map)	name	T/F/U	1,1	F/U	Map	COL	{...}	ϵ	ϵ
column (set)	name	T/F/U	1,1	F/U	Set	COL	{...}	ϵ	ϵ
element of an array	_	T/F/U	1,0/1	F/U	Any-Type	COL	ϵ	ϵ/ trivial /auto-maton	ϵ
element of a tuple	_	T/F/U	1,1	F/U	Any-Type	COL	ϵ/{...}	ϵ/ trivial /auto-maton	ϵ
element of a set	_	T/F/U	1,1	F/U	Any-Type	COL	ϵ/{...}	ϵ/ trivial /auto-maton	ϵ
identifier (simple)	name	T	1,1	T	Simple	COL	ϵ	ϵ	ϵ

4.6 Experiments

MM-infer was implemented not only as a user-friendly tool for interaction with the user during the inference process but also as a tool that enables verification of the correctness and efficiency of the proposed algorithms. In particular, we evaluate RBA against PBA schema inference in terms of execution performance concerning the number of input documents and their structure. The experiments were run over subsets of 6 real-world datasets:

- An 8.192 million record sample of IMDB *title.basics* tab-separated-values (TSV) collection[27] imported into Neo4j graph database.
- A 512 thousand record sample of Wikidata Lexeme namespace JSON collection[28] imported into MongoDB document database.
- A subset of 512 thousand records of Wikidata entities in a single JSON collection dump[29] imported into MongoDB.
- A three collections of Yelp Academic Dataset[30] of JSON documents imported into MongoDB, namely 4.096 million record sample of Review col-

[27] `https://www.imdb.com/interfaces/`
[28] `https://www.wikidata.org/wiki/Wikidata:Database_download`
[29] `https://www.wikidata.org/wiki/Wikidata:Database_download`
[30] `https://www.kaggle.com/datasets/yelp-dataset/yelp-dataset`

Table 4.9: Statistics of the used data sets

Collection	Size (MB)	Average properties	Average nested documents	Average arrays	Maximal document depth	Average document depth
imdb16k	1,19	11,00	1,00	0,00	2	2,00
imdb32k	2,37	11,00	1,00	0,00	2	2,00
imdb64k	4,80	11,00	1,00	0,00	2	2,00
imdb128k	9,63	11,00	1,00	0,00	2	2,00
imdb256k	19,19	11,00	1,00	0,00	2	2,00
imdb512k	39,09	11,00	1,00	0,00	2	2,00
imdb1024k	81,32	11,00	1,00	0,00	2	2,00
imdb2048k	164,17	11,00	1,00	0,00	2	2,00
imdb4096k	331,08	11,00	1,00	0,00	2	2,00
imdb8192k	668,97	11,00	1,00	0,00	2	2,00
lexemes1k	2,09	128,53	39,26	14,73	26	10,92
lexemes2k	3,80	118,33	36,05	13,60	26	10,62
lexemes4k	7,43	115,32	35,06	13,16	26	10,60
lexemes8k	23,99	193,84	56,38	26,29	26	10,10
lexemes16k	109,96	399,25	119,88	50,52	26	13,42
lexemes32k	244,33	439,74	131,86	53,07	26	13,58
lexemes64k	429,01	408,93	120,60	50,95	26	11,93
lexemes128k	863,78	405,33	119,90	48,78	26	12,03
lexemes256k	1560,63	367,57	108,94	45,79	26	11,71
lexemes512k	3202,35	375,31	111,44	46,64	26	11,86
wikidata1k	54,57	3 041,47	825,10	302,71	22	11,96
wikidata2k	99,13	2 753,54	746,45	274,60	22	12,32
wikidata4k	158,38	2 218,33	603,29	221,99	22	12,36
wikidata8k	280,31	1 965,14	533,50	198,65	22	12,55
wikidata16k	492,38	1 738,91	470,97	174,48	22	12,89
wikidata32k	914,44	1 597,41	433,88	158,36	22	12,86
wikidata64k	1611,56	1 398,28	381,05	136,18	22	12,89
wikidata128k	2702,60	1 175,67	321,31	114,15	22	12,91
wikidata256k	4276,11	925,00	255,06	86,86	22	12,73
wikidata512k	6755,81	725,59	201,30	66,62	22	12,60
yelpreview2k	1,41	11,00	1,00	0,00	2	2,00
yelpreview4k	2,77	11,00	1,00	0,00	2	2,00
yelpreview8k	5,57	11,00	1,00	0,00	2	2,00
yelpreview16k	11,12	11,00	1,00	0,00	2	2,00
yelpreview32k	22,27	11,00	1,00	0,00	2	2,00
yelpreview64k	44,58	11,00	1,00	0,00	2	2,00
yelpreview128k	89,48	11,00	1,00	0,00	2	2,00
yelpreview256k	179,15	11,00	1,00	0,00	2	2,00
yelpreview512k	361,99	11,00	1,00	0,00	2	2,00
yelpreview1024k	729,29	11,00	1,00	0,00	2	2,00
yelpreview2048k	1 461,10	11,00	1,00	0,00	2	2,00
yelpreview4096k	2 932,20	11,00	1,00	0,00	2	2,00

Continuation of Table 4.9

Collection	Size (MB)	Average properties	Average nested documents	Average arrays	Maximal document depth	Average document depth
yelptip1k	0,18	7,00	1,00	0,00	2	2,00
yelptip2k	0,37	7,00	1,00	0,00	2	2,00
yelptip4k	0,74	7,00	1,00	0,00	2	2,00
yelptip8k	1,48	7,00	1,00	0,00	2	2,00
yelptip16k	2,95	7,00	1,00	0,00	2	2,00
yelptip32k	5,90	7,00	1,00	0,00	2	2,00
yelptip64k	11,82	7,00	1,00	0,00	2	2,00
yelptip128k	23,68	7,00	1,00	0,00	2	2,00
yelptip256k	47,60	7,00	1,00	0,00	2	2,00
yelptip512k	95,73	7,00	1,00	0,00	2	2,00
yelpuser1k	11,59	24,00	1,00	0,00	2	2,00
yelpuser2k	20,07	24,00	1,00	0,00	2	2,00
yelpuser4k	35,83	24,00	1,00	0,00	2	2,00
yelpuser8k	59,72	24,00	1,00	0,00	2	2,00
yelpuser16k	92,84	24,00	1,00	0,00	2	2,00
yelpuser32k	130,97	24,00	1,00	0,00	2	2,00
yelpuser64k	240,10	24,00	1,00	0,00	2	2,00
yelpuser128k	359,46	24,00	1,00	0,00	2	2,00
yelpuser256k	648,87	24,00	1,00	0,00	2	2,00
yelpuser512k	1 055,76	24,00	1,00	0,00	2	2,00
yelpuser1024k	1 814,98	24,00	1,00	0,00	2	2,00

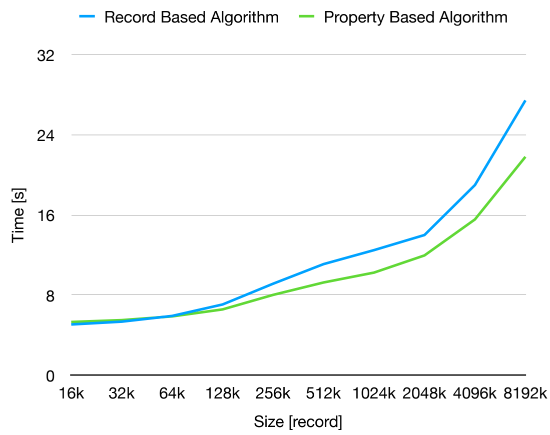
lection, 512 thousand records of Tip collection, and 1.024 million records of User collection.

The characteristics of the selected datasets are listed in Table 4.9 to indicate the growing complexity. Experiments with different data sizes were executed to measure the performance depending on the number and complexity of the input documents.

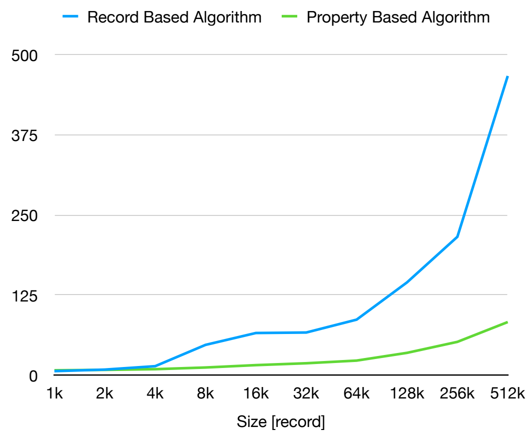
The experiments were performed on a bare-metal virtual machine running on the VMware[31] infrastructure. The allocated hardware resources were CPU Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz (8 core), 64 GB of memory, and a solid-state drive with a capacity of 1.1 TB. Apache Spark was executed locally with 32 GB RAM set to JVM via -Xmx32768M. A possible bias caused by temporary decreases in system resources was mitigated by 20 runs of each algorithm on each extracted subset of input data. The extremes (minimum and maximum) were removed from the measurements, and the remaining measurements were averaged.

Figure 4.16 confirms our hypothesis that the RBA, which utilised the traditional strategy to work with whole records, is slower when run on more complex data (i.e., lexeme or wikidata dump) because it has to work with whole records

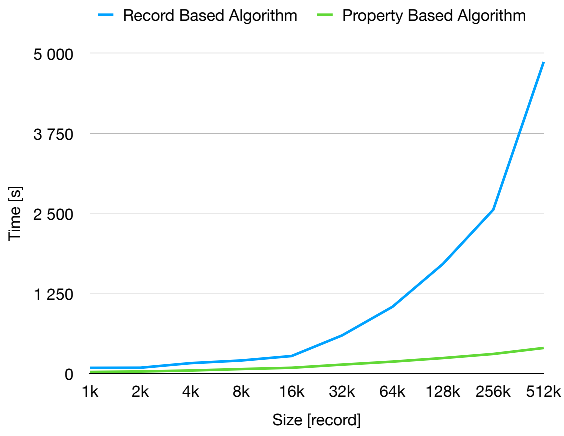
[31] https://www.vmware.com/



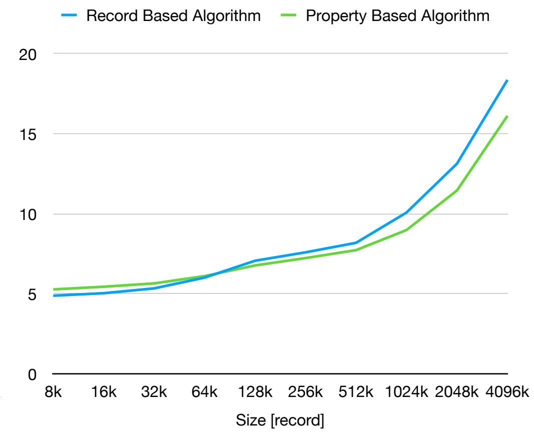
(a) imdb



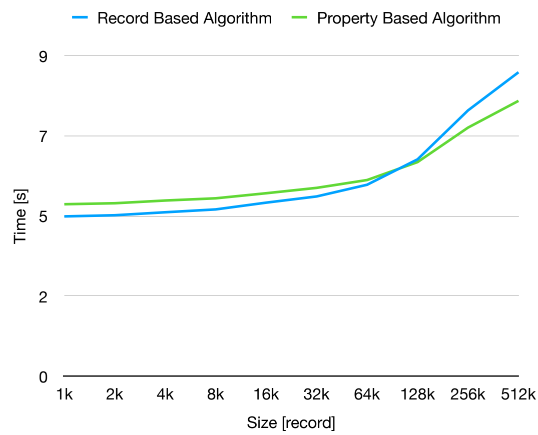
(b) lexemes



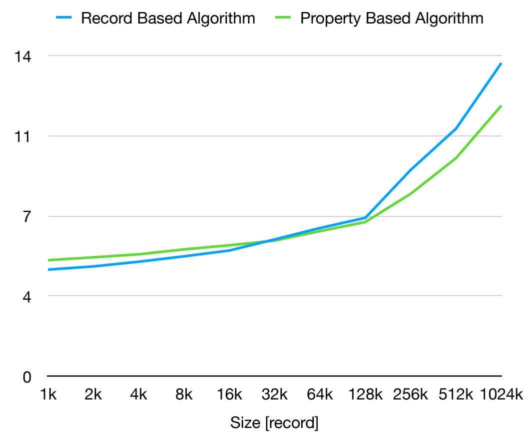
(c) wikidata



(d) yelpreview



(e) yelptip



(f) yelpuser

Figure 4.16: Results of experiments

instead of separate properties. Instead, PBA first merges separate properties and, in the end, it merges them into the resulting schema of the whole kind. In the case of simple data, this difference is not that significant, but larger, more complex documents depict the difference. In addition, PBA is also scalable more easily because it works with the approximately same data portions regardless of the size of the input. Hence, RBA is more suitable for smaller, less complex data. In general, RBA is more appropriate for aggregate-ignorant models with a smaller amount of properties in kinds, whereas PBA is a better choice for aggregate-oriented models.

4.7 Conclusion and Future Work

This paper introduces a novel proposal of an approach dealing with schema inference for multi-model data. Contrary to existing works, it covers all currently popular data models and possible combinations, including cross-model references and data redundancy. In addition, it can cope with large amounts of data – a standard feature in NoSQL databases but not commonly considered in existing schema inference strategies.

The core idea of the proposal is completed, implemented as a tool *MM-infer*, and experimentally verified. Nevertheless, there are still possible directions for extension and exploitation. In our future work, we will focus on the inference of more complex cross-model integrity constraints which can be expressed, e.g., using the Object Constraint Language (OCL) [48]. To infer a more precise target schema, we can also incorporate the eventual knowledge of multi-model queries or semantics of the data. In the former case, we can infer an equivalent schema that reflects the expected data access. In the latter case, we can reveal information that cannot be found in the data itself. Last but not least, the inference approach, together with statistical analysis of the source data, can reveal and enable the backwards correction of errors (i.e., occasionally occurring exceptional cases) in the data.

Paper V

MM-evocat: A Tool for Modelling and Evolution Management of Multi-Model Data

Pavel Koupil[@1], Jáchym Bártík[1], Irena Holubová[1]

Manuscript under review

[@] corresponding author, e-mail: pavel.koupil@matfyz.cuni.cz

[1] Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Abstract

The arrival of so-called *multi-model data* has brought many challenging problems. The contradictory features of the combined models and lack of standardisation of their combination make the solution of the data management specifics highly complex.

In this paper, we focus on the problem of evolution management of multi-model data. With the changing user requirements, the schema and the data need to be adapted to preserve the expected functionality of a multi-model application. We introduce a tool *MM-evocat* based on utilising the category theory. We will show that the core of the tool, i.e., the categorical representation of multi-model data, enables us to grasp all the specifics of the individual models and their possible combinations. Its simple but powerful formal basis enables unique and robust support for evolution management.

Keywords

• Multi-Model Data • Evolution Management • Category Theory

5.1 Introduction

Currently, there exist 388 database management systems (DBMSs).[1] If we consider the 50 most popular representatives, involving the key players, such as, e.g., *Oracle DB*, *PostgreSQL*, *MongoDB*, Microsoft *SQL Database*, *Informix*, etc., we get 60% systems that can be denoted as *multi-model*.

Example 5.1. Let us consider an example of a multi-model scenario as shown in Figure 5.1. The relational data (purple) represents customers and their addresses, whereas the document data (green) is redundant to the relational data. Relationships between customers are captured in the graph (blue). □

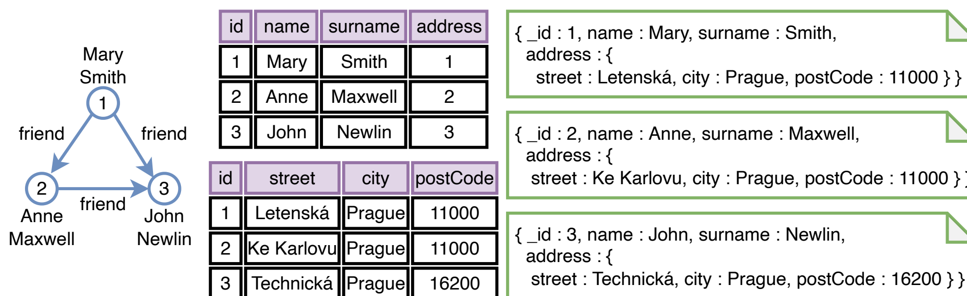


Figure 5.1: An example of multi-model data

According to our extensive survey [13], the features of existing multi-model databases vary significantly. This status is given by the fact that (1) they are based on the distinct original core single model as well as distinct target application domains, (2) there is no acknowledged standard on how to support a

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

combination of models, cross-model querying, multi-model indices, etc., and (3) the combined models have distinct, often contradictory features.

The described variety of multi-model DBMSs lacking standards and the general complexity of multi-model applications bring many new challenges to data management. One of them, our aim in this paper, is *evolution management* of multi-model data. As user requirements or the environment change, the data structures evolve, and the whole system must be adapted to ensure the same functionality and efficiency. We can rely on a skilled database administrator in simpler applications, but in complex cases, where multi-model data undoubtedly belong, it is a complicated and error-prone task.

Example 5.2. Consider again Figure 5.1. We may want to merge customers with their address in a relational model. Or we may want to group the properties *street*, *city*, *postalCode* into a single property *address*. □

To address the indicated problems, we extend our previous research results in evolution management [119, 120] and categorical representation of multi-model data [3]. We introduce a tool called *MM-evocat*,[2] which enables to perform user-specified changes over an abstract multi-model schema and propagates them across all affected sub-models to the logical schemas as well as data instances. The main contributions can be summarised as follows:

- The unifying categorical representation enables us to represent all popular data models (relational, key/value, document, column, and graph) and all types of their combination (embedding, references, and redundancy).
- The representation enables a full decomposition of the data structures to a categorical graph. Hence, it does not impose any initial aggregation of more complex structures, making evolution management robust and efficient.
- Apart from the common and core *schema modification operations* (SMOs), we introduce operations with integrity constraints (ICs), cardinalities, and groups of properties.
- Besides the most natural propagation from the conceptual to the logical level (so-called *heavy operations*), we also introduce *light operations* which change only the conceptual level and its mapping to the logical level.
- We present a proof-of-concept implementation *MM-evocat*. Using real-world data we demonstrate the described features and show how much manual and error-prone work the approach saves.

Note that apart from the data management, the categorical representation has been successfully used in various related areas, such as programming language theory [22, 23], data migration [20], or artificial intelligence (AI) [24, 25].

Paper Outline In Section 2, we review related work. Section 3 introduces the categorical representation of multi-model data, and Section 4 describes the proposed evolution management approach and its implementation *MM-evocat*. In Section 5, we outline its demonstration.

[2] https://www.ksi.mff.cuni.cz/~koupil/mm-evocat/

5.2 Related Work

The problem of evolution management is challenging even in the single-model world. In the case of multi-model approaches, the amount of related work is small and with various limitations.

The evolution management framework *DaemonX* [117] introduces a top-down approach starting from the design of a platform-independent model (PIM) and then mapping its selected parts to the respective single-model platform-specific models (PSMs) followed by schema, operational, and extensional levels. The supported PSM models involve XML, relational, business-process, and REST. The authors focus on the correct and complete propagation of changes. The idea of the framework is general and extensible, so any model mapped to the common general PIM schema can be added to the framework. However, the authors do not consider proper multi-model data because neither cross-model references nor mutual embedding of models is supported.

Another academic prototype system *Darwin* [125, 162] supports several aggregate-oriented NoSQL data stores (i.e., *MongoDB*, *Couchbase*, *Cassandra*, and *ArangoDB*). Beyond extracting a schema summary from a NoSQL data instance, users can derive the historical series of schema versions and semi-automatically declare the changes between two consecutive schemas. *MigCast* [122, 126] extends *Darwin* with an adviser for evaluating different data migration strategies w.r.t. their costs (such as monetary costs for operating in the cloud or the expected overhead in access times).

The approach called *MM-evolver* [120] supports evolution management of both aggregate-oriented and aggregate-ignorant models. In addition, it also supports references, though only in a limited way. However, the level of abstraction of the multi-model data is insufficient, and its further extensions are complicated.

The *ORION language* [163] was designed for evolution management of *U-Schemas* [30], a proposal of representation of multi-model data. Although the authors try to ensure unification, they involve special constructs that cover specific features of particular models. Also, the consideration of inter-model links is limited – the models are handled rather separately, not as one unit. Consequently, the schema modification language is complex, as it must consider all the constructs and their specifics. As in the previous case, further approach extensions will not be straightforward.

The idea to exploit category theory for representation and modification of multi-model data is not new. However, the approaches are still limited. Paper [19] utilises category theory to represent (object-)relational data and considers only basic operation (adding and deletion). The authors of paper [57] utilise category theory to represent multi-model data (in particular relational, JSON document, and graph) and to express the data migration process between the models formally.

5.3 Categorical Conceptual Model

Let us first remind the basic notions of the category theory. A *category* $\mathbf{C} = (\mathcal{O}, \mathcal{M}, \circ)$ consists of a set of objects $\mathcal{O}$, set of morphisms $\mathcal{M}$, and a composition operation $\circ$ over the morphisms. Each morphism is modelled as an arrow f :

$A \to B$, where $A, B \in \mathcal{O}$. We must also ensure *transitivity* ($g \circ f \in \mathcal{M}$ for any $f, g \in \mathcal{M}$, $f : A \to B$, $g : B \to C$), *associativity* (requiring $h \circ (g \circ f) = (h \circ g) \circ f$ for any suitable $f, g, h \in \mathcal{M}$), as well as introduce an *identity* morphism 1_A for each object A such that $f \circ 1_A = f = 1_B \circ f$ for any $f : A \to B$. Category as a whole can be visualised in a form of a multigraph, where objects act as vertices and morphisms as directed edges.

Schema Category The core of conceptual modelling of multi-model data in *MM-evocat* forms so-called *schema category*.[3] To simplify the understanding, we will explain it using the terms known from the ER model,[4] though we do not need to distinguish them, as they are all treated in the same way.

Schema category $\mathbf{S}$ is defined as a tuple $(\mathcal{O}_\mathbf{S}, \mathcal{M}_\mathbf{S}, \circ_\mathbf{S})$. Objects in $\mathcal{O}_\mathbf{S}$ correspond to individual entity types, attributes, and relationship types. Attribute, relationship and hierarchy 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 modelled as a tuple (*key*, *label*, *superid*, *ids*), where $key \in \mathbb{O} \subseteq \mathbb{N}$ is an automatically assigned internal identity, *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 forming the actual data contents of a given object, and $ids \subseteq \mathcal{P}(superid)$, $ids \neq \emptyset$ is a set of particular identifiers (each modelled as a set of attributes) allowing us to uniquely distinguish individual data instances. It holds that $\bigcup_{id \in ids} id \subseteq superid$ (equality for entity or attribute objects).

Each morphism $m \in \mathcal{M}_\mathbf{S}$ is a tuple (*signature*, *dom*, *cod*, *min*, *max*). $signature \in \mathbb{M}^*$ enables to mutually distinguish all morphisms except the identity ones. $\mathbb{M}^*$ is the set of all possible sequences of symbols from $\mathbb{M}$ connected using "$\cdot$", $signature \in \mathbb{M}$ 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, $signature = \varepsilon$ is used for identity. *dom* and *cod* represent the domain and codomain of the morphism. Cardialities $min \in \{\texttt{0}, \texttt{1}\}$ and $max \in \{\texttt{1}, *\}$ allow us to express constraints on minimal/maximal numbers of occurrences. Identity morphism for an object $o \in \mathcal{O}_\mathbf{S}$ is defined as $1_o = (\varepsilon, o, o, \texttt{1}, \texttt{1})$.

For non-identity morphisms $m_1 = (signature_1, dom_1, cod_1, min_1, max_1)$, $m_2 = (signature_2, dom_2, cod_2, min_2, max_2) \in \mathcal{M}_\mathbf{S}$, their composition $m_2 \circ_\mathbf{S} m_1 = (signature_2 \cdot signature_1, dom_1, cod_2, \min(min_1, min_2), \max(max_1, max_2))$.

Example 5.3. Figure 5.2 depicts the ER (a) and categorical (b) conceptual model of sample data from Figure 5.1. □

Mapping To "glue" the categorical conceptual schema with the respective logical schemas in the underlying DBMSs, the schema category is decomposed and mapped to particular model-specific data structures. Since the terminology within the particular popular models differs, in Table 5.1 we provide an overview of the popular models we support, their classification, and the unification of respective model-specific terms.

[3] We also introduced an *instance category* [3] for the unified representation of data instances, but we omit its definition due to space limitations.

[4] We show in [26] that an ER model can be transformed into a schema category.

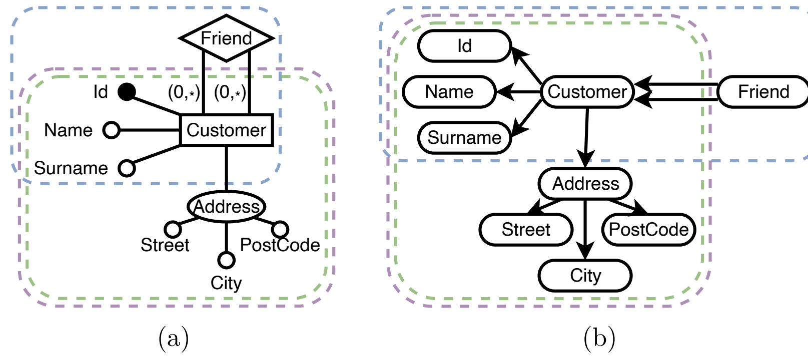


Figure 5.2: ER (a) and categorical (b) schema of sample data

Table 5.1: Unification of terms in popular models

Unifying term	Relational	Graph	Key/value	Document	Column
Kind	Table	Label / type	Bucket	Collection	Column family
Record	Tuple	Node / edge	(key, value)	Document	Row
Property	Attribute	Property	–	Field	Column
Domain	Data type	Data type	–	Data type	Data type
Value	Value	Value	Value	Value	Value
Identifier	Key	Identifier	Key	Identifier	Row key
Reference	Foreign key	–	–	Reference	–
Array	–	Array	Array	Array	Array
Structure	–	–	Set / Zset / Hash / ...	Nested doc. / object	Super column

At the logical level, a transition between two distinct models can be expressed either via (1) *inter-model references* or by (2) *embedding* one model into another (e.g., columns of type JSON in tables of the relational model of *PostgreSQL*[5]). Another possible combination of models is via (3) *multi-model redundancy*, i.e., storing the same data fragment in two or more distinct models.

The decomposition of a schema category **S** (depicted in Figure 5.2 (b) using colours), eventually partial or overlapping, is defined via a set of *mappings*, as also formally defined in [3] (and omitted for paper length). Each mapping describes where and how data instances of a subgraph of **S** (following particular conditions) are stored in a kind of a particular DBMS. For each kind, the mapping specifies the respective DBMS, its name, its root object in **S**, and an *access path* which recursively (i.e., in the same way) describes the structure of a kind, i.e., its (simple or complex) properties, relatively to the root object. The description is rich enough to cover various specifics of the underlying models and their combinations, such as properties with user-defined, anonymous, or dynamically-derived names; properties *inlined* from more distant parts of the categorical graph (via composite morphisms); auxiliary properties used, e.g., for logical grouping of a set of properties; order-preserving/order-ignoring sets of subproperties etc. For easier understanding, we also introduce a JSON-like representation of an access path.

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

Example 5.4. In the following examples, we will depict the mapping only graphically. E.g., in Figure 5.3 the mapping is depicted for kinds (tables) *Address* and *Customer* using the dashed arrows. □

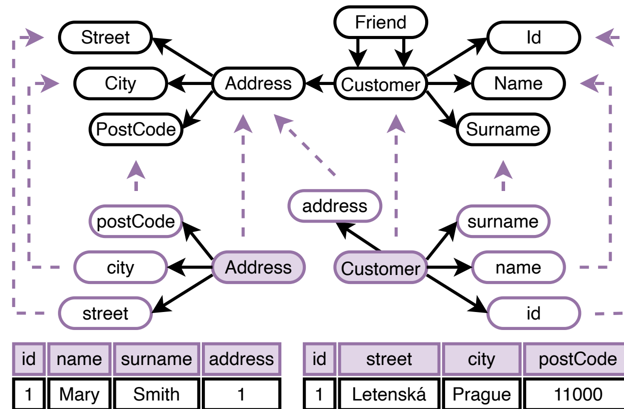


Figure 5.3: Relational-to-conceptual mapping

5.4 Evolution Manager *MM-evocat*

MM-evocat is an extensible modular framework whose core forms the introduced categorical conceptual model. It enables the creation of a categorical model, its decomposition and mapping to any combination of the supported models, and application of user-specified modifications. The architecture of *MM-evocat* is depicted in Figure 5.4. At the lowest, platform-specific (green) level there are the particular DBMSs, each with respective wrappers ensuring the unified communication with the DBMS and unifying representation of the multi-model data. Currently it supports the following models and DBMSs: *PostgreSQL* (relational and document, i.e., multi-model), *Neo4j*[6] (graph), and *MongoDB*[7] (JSON document). The blue logical-to-conceptual mapping represents the “glue” between the logical schemas and conceptual schema at the (red) platform-independent level. Over the conceptual schema, the user can specify evolution/migration operations or conceptual queries. We also proposed an approach for inference of the categorical conceptual schema [109] from the sample data.

In Figure 5.5 we provide a screenshot of *MM-evocat* with the same categorical model as in Figure 5.2.

5.4.1 Schema Modification Operations

For specification of user-required changes, we have defined the *Multi-Model Schema Evolution Language* (MMSEL), i.e., a set of *Schema Modification Operations* (SMOs) that enable to create and delete any allowed schema. Figure 5.6 shows its EBNF grammar.

As we can see, we naturally involve core operations to *add*, *delete*, or *rename* a property. We also support in related work common operations to *copy* or *move*

[6] https://neo4j.com/
[7] https://www.mongodb.com/

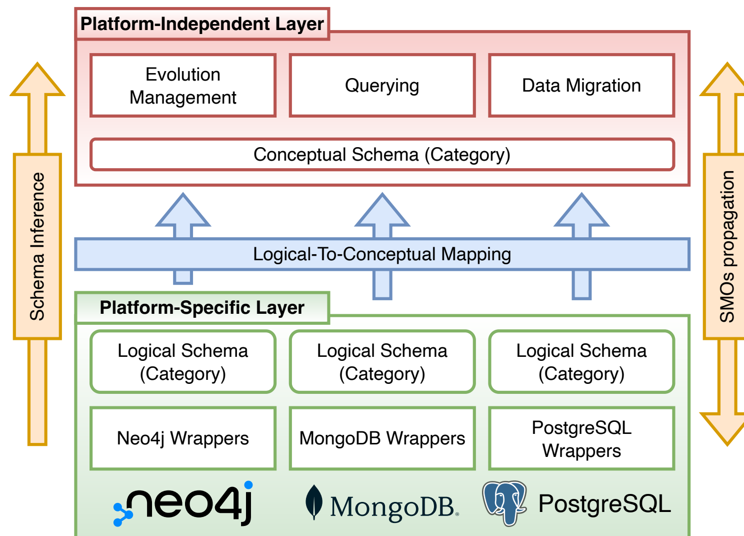


Figure 5.4: Architecture of *MM-evocat*

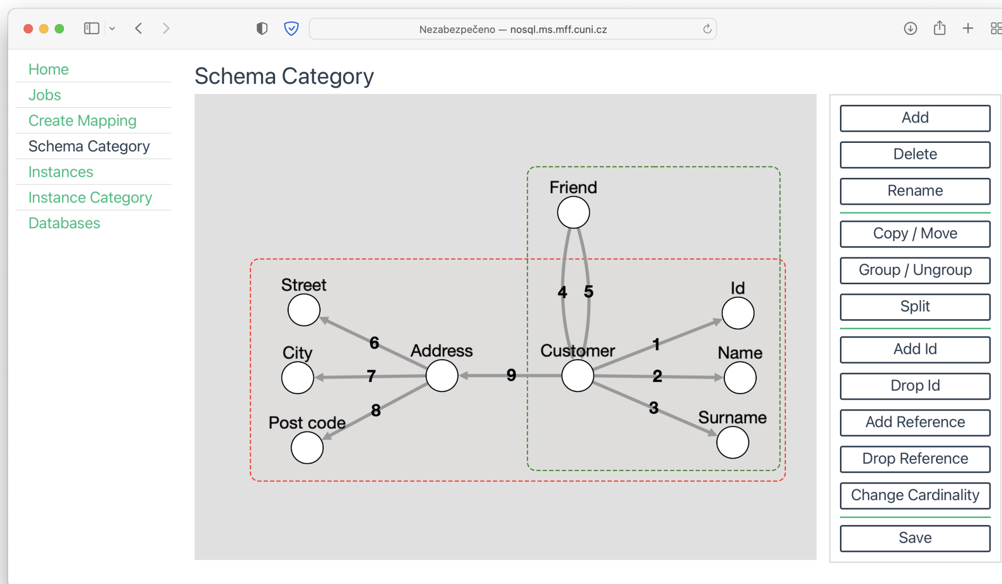


Figure 5.5: A screenshot of *MM-evocat*

a property to another property. We also support the creation of a new property from a set of properties, either by creating a complex parent property (*group*), or by creating a simple property containing their values separated using a specified separator (*union*), including their inverse operations (*ungroup*/*split*). In addition, we enable to work with integrity constraints, namely identifiers (*addId*/*dropId*) and references (*addRef*/*dropRef*). Finally (and also uniquely), we enable the modification of cardinalities in both directions (covering also the change between a simple type and an array), i.e., from more to less restricted and vice versa (*changeCardinality*).[8]

[8]Both directions change the structure of the schema category, similar to the situation of an additional table in case of an M:N relationship.

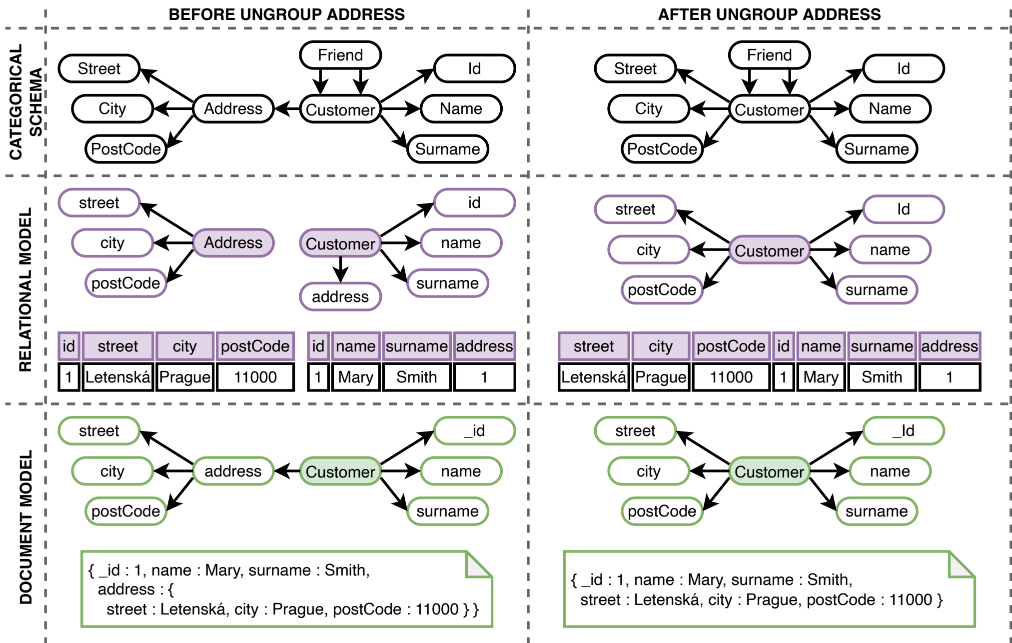
```
MMSEL := ADD | DELETE | RENAME | COPY | MOVE | GROUP | UNGROUP | UNION |
         SPLIT | ADD_ID | DROP_ID | ADD_REF | DROP_REF | CHANGE_CARDINALITY

ADD      := "add"     PROP ["=" {val}] ["to" PROP] [SELECTION]
DELETE   := "delete"  PROP [SELECTION]
RENAME   := "rename"  PROP "to" {name} [SELECTION]
COPY     := "copy"    PROP "to" PROP [SELECTION]
MOVE     := "move"    PROP "to" PROP [SELECTION]
GROUP    := "group"   PROP[","PROP]* "to" PROP [SELECTION]
UNGROUP  := "ungroup" PROP [SELECTION]
UNION    := "union"   PROP[","PROP]* "to" PROP "separator" {sep} [SELECTION]
SPLIT    := "split"   PROP TO PROP[","PROP]* "separator" {sep} [SELECTION]
ADD_ID   := "addId"   PROP[","PROP]* "to" PROP
DROP_ID  := "dropId"  PROP[","PROP]* "from" PROP
ADD_REF  := "addRef"  PROP[","PROP]* "from" PROP "to" PROP
DROP_REF := "dropRef" PROP[","PROP]* "from" PROP "to" PROP
CHANGE_CARDINALITY := "changeCardinality" (PROP1->PROP2 "to" PROP1<-PROP3->PROP2)
    | (PROP1<-PROP3->PROP2 "to" PROP1->PROP2) ["separator" {sep}] [SELECTION]

SELECTION := "where" COND ["and" | "or" COND]*
COND      := PROP "=" {value}
PROP, PROP1, PROP2, PROP3 := [{schemaId}"."{label} | [{schemaId}]"."{key}
<-, ->    := [{schemaId}]"."{signature}
```

Figure 5.6: EBNF syntax of MMSEL

Regarding the propagation, the operations can be further divided into heavy and light, depending on user requirements. A *heavy operation* is an SMO that is propagated from the conceptual level to the mapping, logical level, and data instances. Conversely, a *light operation* only triggers a propagation to the mapping. Each SMO can be heavy. Operations *add*, *rename*, *group*, *addId*, and *addRef* can be also light. (Note that this behaviour is different from the *eager* and *lazy* migration considered in [122, 126]. These classes reflect whether the operation is propagated immediately or the propagation can be delayed until the change is needed.)

Example 5.5. An example of a heavy operation is provided in Figure 5.7. We apply operation *ungroup Address* to extract subproperties *Street*, *City*, and *PostCode* from kind *Address* and to store them directly to kind *Customer*. As we can see, this part of the schema is stored redundantly in both relational (purple) and document (green) models. In the relational model we can see that the respective kinds (tables *Address* and *Customer*) are merged into one kind (table) *Customer*. Conversely, we only remove one level from the hierarchical JSON data in the document model. □

Example 5.6. An example of a light operation is provided in Figure 5.8. We apply operation *group Name, Surname to Personal* to create a new complex property consisting of two existing properties. The change in the categorical schema is propagated to the mapping, however, the logical level remains unchanged. We also provide a simple example of a conceptual query over the schema category which reflects the change at the conceptual level. Conversely, the graph query over the graph (logical) model remains unchanged. □



id	street	city	postCode
1	Letenská	Prague	11000

id	name	surname	address
1	Mary	Smith	1

street	city	postCode	id	name	surname	address
Letenská	Prague	11000	1	Mary	Smith	1

Figure 5.7: Heavy operation *ungroup Address*

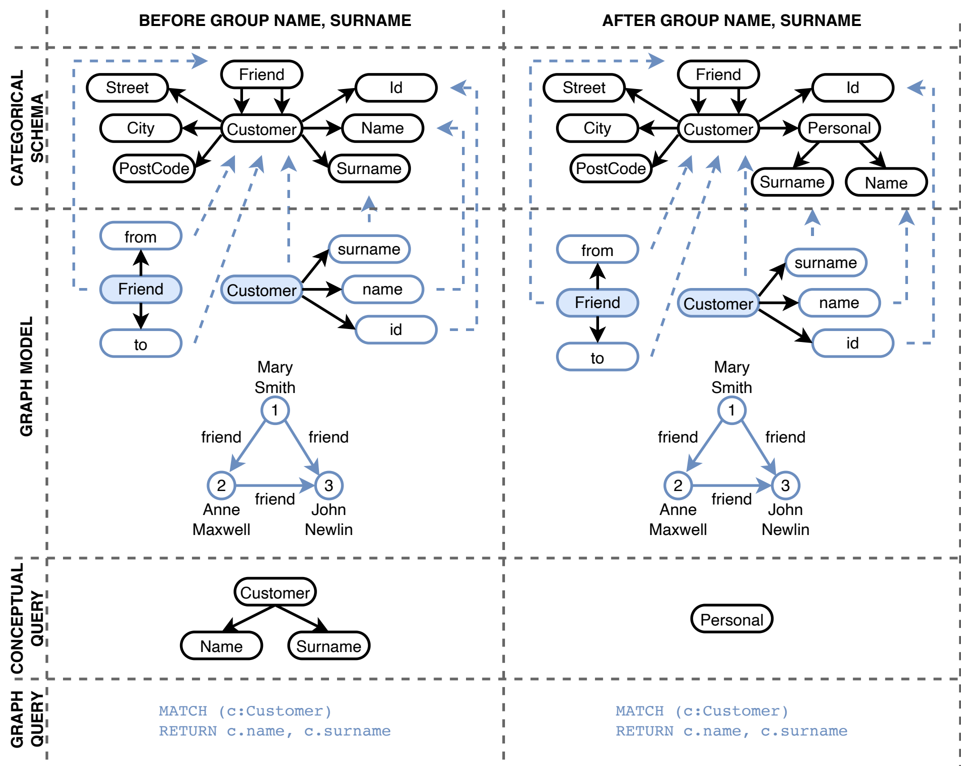


Figure 5.8: Light operation *group Name, Surname to Personal*

5.5 Demonstration Outline

Our demonstration will be based on three kinds of the *Yelp Academic Dataset*[9] of JSON documents, namely the *Review* collection, the *Tip* collection, and the *User*

[9] https://www.kaggle.com/datasets/yelp-dataset/yelp-dataset

collection, transformed into the three supported data models, i.e., graph, JSON document, and relational. Using the data, we will gradually modify the datasets to demonstrate:

1. the advantages of the categorical representation, i.e., its full decomposition, unifying representation of all the models and all types of cross-model transitions etc.,
2. the rich set of SMOs, i.e., the new operations as well as the difference and applicability of light and heavy operations,
3. the number of affected kinds/properties and consequently the amount of saved manual and error-prone work, and
4. the efficient extensibility for the near future work, i.e., the efficient propagation to queries or the utilisation of AI for self-adaptability of the system.

Conclusion

This thesis introduces a general framework for modelling and management of multi-model data. Unlike the existing solutions, the framework is based on a mature formal background of a theory general enough to grasp the variety of all popular data models and to support different data management tasks, such as, e.g., data modelling, schema inference, data migration, and evolution management, in a unified way.

The main contributions are summarised as follows:

- *Unification of data models.* First, an extensive analysis of popular database systems and underlying data models was performed. Based on the results, we proposed a unification of related constructs occurring in various data models and brought them to the same (abstract) level. Hence, contrary to the existing solutions, we do not introduce any constructs tied only to a particular model.

- *Multi-model data modelling.* The proposed data modelling approach is general enough to allow a unified representation of popular data models and their combination at the conceptual level. To verify the completeness of the proposal, an algorithm for translating the ER schema into the proposed categorical representation was proposed. Finally, we provided a unified data representation that serves as a mediator for various data management tasks.

- *The bridge between the conceptual and logical layer.* We have proposed an approach that allows for mapping of the unified categorical schema to any (combination of) popular models supported in existing DBMSs, while the specific features of the logical layer are hidden from the user. Along with the mapping, we introduced data transformation algorithms that, among others, allow to realise data migration between different (combinations of) logical representations. The core idea was implemented in the academic prototype *MM-cat*.

- *Inference of the multi-model schema.* To the best of our knowledge, we have proposed the first approach dealing with the inference of a multi-model schema. In addition, the approach allows to infer a number of integrity constraints, e.g., (simple) identifiers, references (both intra- and inter-model), and to reveal partial and complete redundancy in the data (once again, both intra- and inter-model). Last but not least, exploiting the statistical analysis of the source data, the approach enables the detection and backward correction of errors in the data. The main idea of the proposal was implemented as the academic prototype *MM-infer* and experimentally verified.

- *Evolution management and correct propagation of changes.* Having the unified conceptual layer, schema modifications within and across multiple logical models are reduced to modifications of the unified representation and its mapping to the logical layer. For this purpose, we proposed a several sets of SMOs, together with the respective propagation of changes

via transformation algorithms between the logical and unified layers. The main idea was implemented in the academic prototype *MM-evocat*.

Finally, let us note that the unification of models allows the framework to be applicable to multi-model data represented both polystores and multi-model systems, as well as to single-model systems. Even in the latter case of single-model data the proposed approaches also bring innovations and extensions thanks to the unified and general view of the respective data management tasks.

Current and Future Research

★ Besides extensions to existing components of the framework (see Subsection 3.8.1 ★ and Section 4.7) we are currently working on additional features:

- *Conceptual query language.* The level of abstraction of the categorical approach allows us to define a conceptual query language. Utilising the ideas of decomposition and mapping, any conceptual expression can be decomposed and further translated into particular query expressions at the logical level. Moreover, since a category can be seen as a special type of a multigraph, the categorical query language can be inspired by graph pattern matching, as known from existing graph languages such as, e.g., Cypher, and SPARQL.

- *Conceptual query evaluation plan.* The knowledge of a unifying schema and its decomposition allows the construction of multiple query evaluation strategies. Similar to single-model systems, there is an opportunity for creating multiple query execution plans and selecting the optimal query evaluation strategy. Moreover, we can exploit the natural properties of multi-model data, e.g., cross-model redundancy in the data, which allows for higher variability in query evaluation strategies.

- *Self-Adapting Evolution Management.* The combination of the variety of data formats and the continuous changes in the data bring a huge challenge for administrators of (not only multi-model) database systems. Our future goal is first to extend the existing evolution management proposal with the propagation of changes to queries and then to focus on the area of autonomous management of rapidly changing multi-model Big Data, as envisioned in our paper [164].

Bibliography

[1] Irena Holubova, Pavel Contos, and Martin Svoboda. Categorical Management of Multi-Model Data. In *25th International Database Engineering & Applications Symposium*, Ideas 2021, page 134–140, New York, NY, USA, 2021. Association for Computing Machinery.

[2] Martin Svoboda, Pavel Čontoš, and Irena Holubová. Categorical Modeling of Multi-Model Data: One Model to Rule Them All. In *Model and Data Engineering*, Lecture Notes in Computer Science, pages 190–198, Cham, 2021. Springer International Publishing.

[3] Pavel Koupil and Irena Holubová. A unified representation and transformation of multi-model data using category theory. *J. Big Data*, 9(1):1–49, 2022.

[4] Pavel Koupil, Sebastián Hricko, and Irena Holubová. A Universal Approach for Multi-Model Schema Inference. *J. Big Data (accepted)*, 2022.

[5] Pavel Koupil, Jáchym Bártík, and Irena Holubová. MM-evocat: A Tool for Modelling and Evolution Management of Multi-Model Data.

[6] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, 1970.

[7] Pramod J Sadalage and Martin Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2013.

[8] Boyan Kolev, Raquel Pau, Oleksandra Levchenko, Patrick Valduriez, Ricardo Jiménez-Peris, and José Orlando Pereira. Benchmarking Polystores: The CloudMdSQL Experience. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2574–2579, New York, NY, USA, 2016. IEEE.

[9] Jennie Duggan, Aaron J Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. The bigdawg polystore system. *ACM Sigmod Record*, 44(2):11–16, 2015.

[10] Rana Alotaibi, Bogdan Cautis, Alin Deutsch, Moustafa Latrache, Ioana Manolescu, and Yifei Yang. ESTOCADA: towards scalable polystore systems. *Proceedings of the VLDB Endowment*, 13(12):2949–2952, 2020.

[11] Jiaheng Lu, Irena Holubová, and Bogdan Cautis. Multi-model Databases and Tightly Integrated Polystores: Current Practices, Comparisons, and Open Challenges. In *Proc. of CIKM '18*, pages 2301–2302. Acm, 2018.

[12] Jiaheng Lu, Zhen Hua Liu, Pengfei Xu, and Chao Zhang. UDBMS: Road to Unification for Multi-model Data Management. In *ER '18 Workshops*, volume 11158 of *Lecture Notes in Computer Science*, pages 285–294, Cham, 2018. Springer.

[13] Jiaheng Lu and Irena Holubová. Multi-Model Databases: A New Journey to Handle the Variety of Data. *ACM Computing Surveys*, 52(3), 2019.

[14] Irena Holubová, Martin Svoboda, and Jiaheng Lu. Unified Management of Multi-model Data. In *Proc. of ER '19*, pages 439–447. Springer, 2019.

[15] Heikki Mannila and Kari-Jouko Räihä. *The design of relational databases.* Addison-Wesley Longman Publishing Co., Inc., 1992.

[16] Pavel Contos and Martin Svoboda. JSON Schema Inference Approaches. In Georg Grossmann and Sudha Ram, editors, *Advances in Conceptual Modeling - ER 2020 Workshops CMAI, CMLS, CMOMM4FAIR, CoMoNoS, EmpER, Vienna, Austria, November 3-6, 2020, Proceedings*, volume 12584 of *Lecture Notes in Computer Science*, pages 173–183. Springer, 2020.

[17] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

[18] Lippe, E. and Ter Hofstede, A. H. M. A Category Theory Approach to Conceptual Data Modeling. *RAIRO Theor. Informatics Appl.*, 30(1):31–79, 1996.

[19] Chris Tuijn and Marc Gyssens. CGOOD, a Categorical Graph-oriented Object Data Model. *Theoretical Computer Science*, 160(1):217–239, 1996.

[20] David I Spivak and Ryan Wisnesky. Relational Foundations for Functorial Data Migration. In *Proceedings of the 15th Symposium on Database Programming Languages*, Dbpl 2015, pages 21–28, New York, NY, USA, 2015. Association for Computing Machinery.

[21] Joshua Shinavier and Ryan Wisnesky. Algebraic property graphs. *arXiv preprint arXiv:1909.04881*, 2019.

[22] Benjamin C Pierce. *Basic Category Theory for Computer Scientists.* MIT press, 1991.

[23] F William Lawvere and Stephen H Schanuel. *Conceptual Mathematics: a First Introduction to Categories.* Cambridge University Press, 2009.

[24] Michael J Healy, Richard D Olinger, Robert J Young, Shawn E Taylor, Thomas Caudell, and Kurt W Larson. Applying Category Theory to Improve the Performance of a Neural Architecture. *Neurocomputing*, 72(13-15):3158–3173, 2009.

[25] Dan Shiebler, Bruno Gavranović, and Paul Wilson. Category Theory in Machine Learning. *arXiv preprint arXiv:2106.07032*, 2021.

[26] Irena Holubova, Pavel Contos, and Martin Svoboda. Multi-Model Data Modeling and Representation: State of the Art and Research Challenges. In *25th International Database Engineering & Applications Symposium*, Ideas 2021, page 242–251, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Paolo Atzeni, Francesca Bugiotti, Luca Cabibbo, and Riccardo Torlone. Data Modeling in the NoSQL World. *Computer Standards & Interfaces*, 67:103–149, 2020.

[28] Jeremy Kepner, Julian Chaidez, Vijay Gadepally, and Hayden Jansen. Associative Arrays: Unified Mathematics for Spreadsheets, Databases, Matrices, and Graphs. *CoRR*, abs/1501.05709, 2015.

[29] Eric Leclercq and Marinette Savonnet. TDM: A Tensor Data Model for Logical Data Independence in Polystore Systems. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, pages 39–56, Cham, 2019. Springer International Publishing.

[30] Carlos J. Fernández Candel, Diego Sevilla Ruiz, and Jesús J. García-Molina. A unified metamodel for NoSQL and relational databases. *Information Systems*, 104:101898, 2022.

[31] P.P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[32] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified modeling language reference manual*. Pearson Higher Education, 2004.

[33] Arthur HM ter Hofstede, Ernst Lippe, and Paul J. M. Frederiks. Conceptual data modelling from a categorical perspective. *The Computer Journal*, 39(3):215–231, 1996.

[34] Jean-Jacques VR Wintraecken. *The NIAM information analysis method: theory and practice*. Springer Science & Business Media, 2012.

[35] Terry A Halpin and Maria E Orlowska. Fact-oriented modelling for data analysis. *Information Systems Journal*, 2(2):97–119, 1992.

[36] Terry Halpin. Fact-oriented modeling: Past, present and future. In *Conceptual modelling in information systems engineering*, pages 19–38. Springer, 2007.

[37] Richard Hull and Roger King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys (CSUR)*, 19(3):201–260, 1987.

[38] L Kerschberg and JES Pacheco. A Functional Data Base Model, Technical Report, 1976.

[39] David W Shipman. The functional data model and the data languages DAPLEX. *ACM Transactions on Database Systems (TODS)*, 6(1):140–173, 1981.

[40] Michael Hammer and Dennis Mc Leod. Database description with SDM: A semantic database model. *ACM Transactions on Database Systems (TODS)*, 6(3):351–386, 1981.

[41] Serge Abiteboul and Richard Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems (TODS)*, 12(4):525–565, 1987.

[42] Jaroslav Pokorný and Karel Richta. Towards Conceptual and Logical Modelling of NoSQL Databases. In Emilio Insfran, Fernando González, Silvia Abrahão, Marta Fernández, Chris Barry, Michael Lang, Henry Linger, and Christoph Schneider, editors, *Advances in Information Systems Development*, pages 255–272, Cham, 2022. Springer International Publishing.

[43] Il-Yeol Song, Mary Evans, and Eun K Park. A comparative analysis of entity-relationship diagrams. *Journal of Computer and Software Engineering*, 3(4):427–459, 1995.

[44] David S. Reiner, Michael L. Brodie, Gretchen Brown, Mark Friedell, David Kramlich, John Lehman, and Arnon Rosenthal. The Database Design and Evaluation Workbench (DDEW) Project at CCA. *IEEE Database Eng. Bull.*, 7(4):10–15, 1984.

[45] Toby J Teorey. *Database modeling and design*. Morgan Kaufmann, 1999.

[46] Jeff Hoffer, Ramesh Venkataraman, and Heikki Topi. *Modern database management*. Pearson Education Limited, 2016.

[47] Thomas A Bruce. *Designing quality databases with IDEF1X information models*. Dorset House Publishing Co., Inc., 1992.

[48] Object Management Group. Object Constraint Language (OCL), version 2.4, 2014.

[49] Martin Necasky. XSEM: A Conceptual Model for XML. In *Proc. of APCCM '07 - Volume 67*, page 37–48, Aus, 2007. ACS, Inc.

[50] Marc Gyssens, Jan Paredaens, and Dirk Van Gucht. A graph-oriented object model for database end-user interfaces. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of data*, pages 24–33, 1990.

[51] Tomáš Hruška and Petr Kolenčík. Comparison of categorical foundations of object-oriented database model. In *International Conference on Deductive and Object-Oriented Databases*, pages 302–319. Springer, 1997.

[52] Zinovy Diskin and Boris Cadish. Variable sets and functions framework for conceptual modeling: Integrating ER and OO via sketches with dynamic markers. In *International Conference on Conceptual Modeling*, pages 226–237. Springer, 1995.

[53] Zinovy Diskin. Generalized sketches as an algebraic graph-based framework for semantic modeling and database design. *University of Latvia: Riga, Latvia*, 1997.

[54] Kristopher S Brown, David I Spivak, and Ryan Wisnesky. Categorical data integration for computational science. *Computational Materials Science*, 164:127–132, 2019.

[55] David I. Spivak. Functorial data migration. *Information and Computation*, 217:31–51, 2012.

[56] Zhen Hua Liu, Jiaheng Lu, Dieter Gawlick, Heli Helskyaho, Gregory Pogossiants, and Zhe Wu. Multi-model Database Management Systems - A Look Forward. In *VLDB '18 Workshops*, pages 16–29. Springer, 2019.

[57] Valter Uotila and Jiaheng Lu. A formal category theoretical framework for multi-model data transformations. In El Kindi Rezig, Vijay Gadepally, Timothy G. Mattson, Michael Stonebraker, Tim Kraska, Fusheng Wang, Gang Luo, Jun Kong, and Alevtina Dubovitskaya, editors, *Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB Workshops, Poly 2021 and DMAH 2021, Virtual Event, August 20, 2021, Revised Selected Papers*, volume 12921 of *Lecture Notes in Computer Science*, pages 14–28. Springer, 2021.

[58] Patrick Schultz, David I. Spivak, and Ryan Wisnesky. Algebraic Model Management: A Survey. In Phillip James and Markus Roggenbach, editors, *Recent Trends in Algebraic Development Techniques*, pages 56–69, Cham, 2017. Springer International Publishing.

[59] Ralph Abraham, Jerrold E Marsden, and Tudor Ratiu. *Manifolds, tensor analysis, and applications*, volume 75. Springer Science & Business Media, 2012.

[60] Alberto Hernández Chillón, Diego Sevilla Ruiz, and Jesús García Molina. Athena: A database-independent schema definition language. In Iris Reinhartz-Berger and Shazia W. Sadiq, editors, *Advances in Conceptual Modeling - ER 2021 Workshops CoMoNoS, EmpER, CMLS, St. John's, NL, Canada, October 18-21, 2021, Proceedings*, volume 13012 of *Lecture Notes in Computer Science*, pages 33–42. Springer, 2021.

[61] Kristóf Marussy, Oszkár Semeráth, Aren A Babikian, and Dániel Varró. A Specification Language for Consistent Model Generation based on Partial Models. *J. Object Technol.*, 19(3):3–1, 2020.

[62] Franz Baader, Diego Calvanese, Deborah McGuinness, Peter Patel-Schneider, Daniele Nardi, et al. *The description logic handbook: Theory, implementation and applications.* Cambridge university press, 2003.

[63] Pavel Contos. Abstract model for multi-model data. In Christian S. Jensen, Ee-Peng Lim, De-Nian Yang, Wang-Chien Lee, Vincent S. Tseng, Vana Kalogeraki, Jen-Wei Huang, and Chih-Ya Shen, editors, *Database Systems for Advanced Applications - 26th International Conference, DASFAA 2021, Taipei, Taiwan, April 11-14, 2021, Proceedings, Part III*, volume 12683 of *Lecture Notes in Computer Science*, pages 647–651. Springer, 2021.

[64] Pavel Koupil, Martin Svoboda, and Irena Holubova. 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)*, 2021, pages 635–639, New York, NY, USA, 2021. IEEE.

[65] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Parametric schema inference for massive JSON datasets. *The VLDB Journal*, 2019.

[66] D. Beckett. *RDF/XML Syntax Specification (Revised)*. W3c, 2004.

[67] Hanâ Lbath, Angela Bonifati, and Russ Harmer. Schema Inference for Property Graphs. In Yannis Velegrakis, Demetris Zeinalipour-Yazti, Panos K. Chrysanthis, and Francesco Guerra, editors, *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, pages 499–504. OpenProceedings.org, 2021.

[68] Enrico Gallinucci, Matteo Golfarelli, Stefano Rizzi, Alberto Abelló, and Oscar Romero. Interactive multidimensional modeling of linked data for exploratory OLAP. *Inf. Syst.*, 77:86–104, 2018.

[69] Redouane Bouhamoum, Kenza Kellou-Menouer, Stephane Lopes, and Zoubida Kedad. Scaling up Schema Discovery for RDF Datasets. In *2018 IEEE Icdew*, pages 84–89. IEEE, 2018.

[70] Michael DiScala and Daniel J Abadi. Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data. In *Sigmod '16*, pages 295–310, 2016.

[71] W3c. Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008.

[72] Irena Mlýnková and Martin Nečaský. Heuristic Methods for Inference of XML Schemas: Lessons Learned and Open Issues. *Informatica*, 24(4):577–602, 2013.

[73] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. Inference of Concise Regular Expressions and DTDs. *ACM Trans. Database Syst.*, 35(2):11:1–11:47, 2010.

[74] K. E. Shafer. Creating DTDs via the GB-Engine and Fred. In *Sgml'95*, page 399. Graphic Communications Association, 1995. http://xml.coverpages.org/shaferGB.html.

[75] Chuang-Hue Moh, Ee-Peng Lim, and Wee-Keong Ng. Re-engineering Structures from Web Documents. In *Proceedings of the fifth ACM Conference on Digital libraries*, Dl '00, pages 67–76, San Antonio, Texas, United States, 2000. ACM, New York, NY, USA.

[76] R. K. Wong and J. Sankey. *On Structural Inference for XML Data*. Report UNSW-CSE-TR-0313, School of Computer Science, The University of New South Wales, 2003.

[77] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: a System for Extracting Document Type Descriptors from XML Documents. *SIGMOD Rec.*, 29:165–176, 2000.

[78] Ondřej Vošta, Irena Mlýnková, and Jaroslav Pokorný. Even an Ant Can Create an XSD. In *Database Systems for Advanced Applications*, volume 4947 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2008.

[79] Boris Chidlovskii. Schema Extraction from XML Collections. In *Proceedings of the 2nd ACM/IEEE-CS Joint Conference on Digital libraries*, Jcdl '02, pages 291–292, Portland, Oregon, USA, 2002. ACM, New York, NY, USA.

[80] H. Ahonen. *Generating Grammars for Structured Documents Using Grammatical Inference Methods.* Report A-1996-4, Department of Computer Science, University of Helsinki, 1996.

[81] Henning Fernau. Learning XML Grammars. In Petra Perner, editor, *Machine Learning and Data Mining in Pattern Recognition*, volume 2123 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 2001.

[82] Jun-Ki Min, Jae-Yong Ahn, and Chin-Wan Chung. Efficient Extraction of Schemas for XML Documents. *Inf. Process. Lett.*, 85:7–12, 2003.

[83] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of Concise DTDs from XML Data. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, Vldb '06, pages 115–126, Seoul, Korea, 2006. VLDB Endowment.

[84] Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. Learning Deterministic Regular Expressions for the Inference of Schemas from XML Data. *ACM Transactions on the Web*, 4(4):14:1–14:32, 2010.

[85] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of Concise Regular Expressions and DTDs. *ACM Trans. Database Syst.*, 35(2):11:1–11:47, 2010.

[86] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring XML Schema Definitions from XML Data. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, Vldb '07, pages 998–1009, Vienna, Austria, 2007. VLDB Endowment.

[87] E. M. Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, 1967.

[88] Julie Vyhnanovská and Irena Mlýnková. Interactive Inference of XML Schemas. In *Research Challenges in Information Science (RCIS), 2010 Fourth International Conference on*, pages 191–202. IEEE Computer Society Press, May 2010.

[89] Angelo Augusto Frozza, Ronaldo dos Santos Mello, and Felipe de Souza da Costa. An Approach for Schema Extraction of JSON and Extended JSON Document Collections. In *Iri 2018*, pages 356–363. IEEE, 2018.

[90] Severino Feliciano Morales. *Inferring NoSQL Data Schemas with Model-Driven Engineering Techniques.* PhD thesis, University of Murcia, Murcia, Spain, March 2017.

[91] Ivan Veinhardt Latták. and Pavel Koupil. A Comparative Analysis of JSON Schema Inference Algorithms. In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE,*, pages 379–386. Insticc, SciTePress, 2022.

[92] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. Inferring Versioned Schemas from NoSQL Databases and Its Applications. In *Conceptual Modeling*, pages 467–480. Springer, 2015.

[93] Alberto Hernández Chillón, Severino Feliciano Morales, Diego Sevilla, and Jesús García Molina. Exploring the Visualization of Schemas for Aggregate-Oriented NoSQL Databases. In *Proceedings of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th International Conference on Conceptual Modelling (ER 2017), Valencia, Spain, - November 6-9, 2017.*, pages 72–85, 2017.

[94] Meike Klettke, Uta Störl, and Stefanie Scherzinger. Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, pages 425–444, 2015.

[95] Meike Klettke, Hannes Awolin, Uta Storl, Daniel Muller, and Stefanie Scherzinger. Uncovering the Evolution History of Data Lakes. In *2017 IEEE International Conference on Big Data*, pages 2380–2389, New York, United States, 2017. IEEE.

[96] Mark Lukas Möller, Nicolas Berton, Meike Klettke, Stefanie Scherzinger, and Uta Störl. jhound: Large-scale profiling of open JSON data. *Btw 2019*, 2019.

[97] Michael Fruth, Kai Dauberschmidt, and Stefanie Scherzinger. Josch: Managing Schemas for NoSQL Document Stores. In *Icde '21*, pages 2693–2696. IEEE, 2021.

[98] Javier Luis Cánovas Izquierdo and Jordi Cabot. Discovering Implicit Schemas in JSON Data. In *Icwe '13*, pages 68–83. Springer, 2013.

[99] Javier Luis Cánovas Izquierdo and Jordi Cabot. JSONDiscoverer: Visualizing the Schema Lurking behind JSON Documents. *Knowledge-Based Systems*, 103:52–55, 2016.

[100] Lanjun Wang, Shuo Zhang, Juwei Shi, Limei Jiao, Oktie Hassanzadeh, Jia Zou, and Chen Wangz. Schema Management for Document Stores. *Proc. VLDB Endow.*, 8(9):922–933, 2015.

[101] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON Schema. In *Proceedings of the 25th International Conference on World Wide Web*, page 263–273, 2016.

[102] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. A Type System for Interactive JSON Schema Inference. In *Icalp 2019*, volume 132 of *LIPIcs*, pages 101:1–101:13, 2019.

[103] Angelo Augusto Frozza, Eduardo Dias Defreyn, and Ronaldo dos Santos Mello. A Process for Inference of Columnar NoSQL Database Schemas. In *Anais do XXXV Simpósio Brasileiro de Bancos de Dados*, pages 175–180. Sbc, 2020.

[104] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures Second Edition. World Wide Web Consortium, Recommendation REC-xmlschema-1-20041028, October 2004.

[105] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition, W3C Recommendation, October 2004. http://www.w3.org/TR/xmlschema-2/.

[106] JSON Schema – Specification, 2021.

[107] Ivan Veinhardt Latták. *Schema Inference for NoSQL Databases.* Master thesis, Charles University in Prague, Czech Republic, 2021.

[108] Burton H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.

[109] Pavel Koupil, Sebastián Hricko, and Irena Holubová. MM-infer: A Tool for Inference of Multi-Model Schemas. In Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang, editors, *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, pages 2:566–2:569. OpenProceedings.org, 2022.

[110] Uta Störl and Meike Klettke. Darwin: A Data Platform for NoSQL Schema Evolution Management and Data Migration. In *Workshop Proceedings of the EDBT/ICDT 2022 Joint Conference (March 29-April 1, 2022), Edinburgh, UK*, 2022.

[111] Mark Lukas Möller, Meike Klettke, and Uta Störl. EvoBench - A Framework for Benchmarking Schema Evolution in NoSQL. In Xintao Wu, Chris Jermaine, Li Xiong, Xiaohua Hu, Olivera Kotevska, Siyuan Lu, Weija Xu, Srinivas Aluru, Chengxiang Zhai, Eyhab Al-Masri, Zhiyuan Chen, and Jeff Saltz, editors, *2020 IEEE International Conference on Big Data (IEEE BigData 2020), Atlanta, GA, USA, December 10-13, 2020*, pages 1974–1984. IEEE, 2020.

[112] Jiaheng Lu. Towards Benchmarking Multi-Model Databases. In *CIDR 2017*. www.cidrdb.org, 2017.

[113] Chao Zhang and Jiaheng Lu. Holistic evaluation in multi-model databases benchmarking. *Distributed and Parallel Databases*, 39:1–33, 2019.

[114] Carlo A. Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *Proc. VLDB Endow.*, 4(2):117–128, nov 2010.

[115] Souvik Bhattacherjee, Gang Liao, Michael Hicks, and Daniel J. Abadi. BullFrog: Online Schema Evolution via Lazy Evaluation. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 194–206, New York, NY, USA, 2021. Association for Computing Machinery.

[116] Alberto Hernández Chillón, Meike Klettke, Diego Sevilla Ruiz, and Jesús García Molina. A Taxonomy of Schema Changes for NoSQL Databases. *arXiv preprint arXiv:2205.11660*, 2022.

[117] Marek Polák, Martin Nečaský, and Irena Holubová. DaemonX: Design, Adaptation, Evolution, and Management of Native XML (and More Other) Formats. In *Iiwas '13*, pages 484–493, New York, USA, 2013. Acm.

[118] Jakub Klímek, Jakub Malỳ, Martin Nečaskỳ, and Irena Holubová. eXolutio: methodology for design and evolution of XML schemas using conceptual modeling. *Informatica*, 26(3):453–472, 2015.

[119] M. Nečaský, J. Klímek, J. Malý, and I. Mlýnková. Evolution and Change Management of XML-based Systems. *Elsevier*, 85(3):683–707, 2012.

[120] Irena Holubová, Michal Vavrek, and Stefanie Scherzinger. Evolution Management in Multi-Model Databases. *Data Knowl. Eng.*, 136:101932, 2021.

[121] Jérôme Fink, Maxime Gobert, and Anthony Cleve. Adapting Queries to Database Schema Changes in Hybrid Polystores. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 28 - October 2, 2020*, pages 127–131. IEEE, 2020.

[122] Andrea Hillenbrand, Maksym Levchenko, Uta Störl, Stefanie Scherzinger, and Meike Klettke. MigCast: Putting a Price Tag on Data Model Evolution in NoSQL Data Stores. In *Sigmod '19*, page 1925–1928. Acm, 2019.

[123] Steve Awodey. *Category Theory*. Oxford university press, 2010.

[124] David I Spivak. Category theory for scientists. *arXiv preprint arXiv:1302.6946*, 2013.

[125] Uta Störl, Daniel Müller, Alexander Tekleab, Stephane Tolale, Julian Stenzel, Meike Klettke, and Stefanie Scherzinger. Curating Variational Data in Application Development. In *Proc. of ICDE '18*, pages 1605–1608, 2018.

[126] Andrea Hillenbrand, Uta Störl, Maksym Levchenko, Shamil Nabiyev, and Meike Klettke. Towards Self-Adapting Data Migration in the Context of Schema Evolution in NoSQL Databases. In *ICDE Workshops 2020*, pages 133–138. IEEE, 2020.

[127] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Managing schema evolution in NoSQL data stores. *arXiv preprint arXiv:1308.0514*, 2013.

[128] Andrea Hillenbrand, Uta Störl, Shamil Nabiyev, and Meike Klettke. Self-adapting data migration in the context of schema evolution in NoSQL databases. *Distributed and Parallel Databases*, 40(1):5–25, 2022.

[129] Andrea Hillenbrand, Stefanie Scherzinger, and Uta Störl. Remaining in Control of the Impact of Schema Evolution in NoSQL Databases. In *International Conference on Conceptual Modeling*, pages 149–159. Springer, 2021.

[130] Meike Klettke, Uta Störl, Manuel Shenavai, and Stefanie Scherzinger. NoSQL Schema Evolution and Big Data Migration at Scale. In *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, pages 2764–2774. IEEE, 2016.

[131] Uta Störl, Alexander Tekleab, Meike Klettke, and Stefanie Scherzinger. In for a Surprise When Migrating NoSQL Data. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, page 1662. IEEE Computer Society, 2018.

[132] Mark Lukas Möller, Meike Klettke, Andrea Hillenbrand, and Uta Störl. Query Rewriting for Continuously Evolving NoSQL Databases. In *Conceptual Modeling - 38th International Conference, ER 2019, Salvador, Brazil, November 4-7, 2019, Proceedings*, volume 11788 of *Lncs*, pages 213–221. Springer, 2019.

[133] Michael Barr and Charles Wells. *Category Theory for Computing Science*, volume 1. Prentice Hall, New York, 1990.

[134] Patrick Schultz, David I. Spivak, Christina Vasilakopoulou, and Ryan Wisnesky. Algebraic Databases. *Theory & Applications of Categories*, 32(16-19):547–619, 2017.

[135] Laurent Thiry, Heng Zhao, and Michel Hassenforder. Categories for (Big) Data Models and Optimization. *Journal of Big Data*, 5(1):1–20, 2018.

[136] Atzeni Paolo, Stefano Ceri, Stefano Paraboschi, and Riccardo Torlone. Database Systems: Concepts, Languages and Architectures, 1999.

[137] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Alfonso Pierantonio, and Ludovico Iovino. 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*, pages 1–5, New York, NY, USA, 2020. Association for Computing Machinery.

[138] M. Kolonko and S. Müllenbach. Polyglot Persistence in Conceptual Modeling for Information Analysis. In *2020 10th International Conference on Advanced Computer Information Technologies (ACIT)*, pages 590–594, New York, NY, USA, 2020. IEEE.

[139] Suad Alagić and Philip A. Bernstein. A Model Theory for Generic Schema Management. In *Database Programming Languages*, pages 228–246. Springer, 2002.

[140] Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. UniBench: A Benchmark for Multi-model Database Management Systems. In *Technology Conference on Performance Evaluation and Benchmarking '18*, volume 11135 of *Lecture Notes in Computer Science*, pages 7–23, Cham, 2018. Springer.

[141] Qingsong Guo, Jiaheng Lu, Chao Zhang, Calvin Sun, and Steven Yuan. Multi-Model Data Query Languages and Processing Paradigms. In *Cikm '20*, pages 3505–3506. Acm, 2020.

[142] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases. *CoRR*, abs/1405.3631, 2014.

[143] Michal Vavrek, Irena Holubová, and Stefanie Scherzinger. MM-evolver: A Multi-model Evolution Management Tool. In *Edbt '19*, pages 586–589. OpenProceedings.org, 2019.

[144] Michael Stonebraker and Ugur Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Icde 2005*, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[145] Donald Feinberg, Merv Adrian, Nick Heudecker, Adam M. Ronthal, and Terilyn Palanca. Gartner Magic Quadrant for Operational Database Management Systems, 2015.

[146] Jiaheng Lu and Irena Holubová. Multi-model Data Management: What's New and What's Next? In *Proceeding of the 20th International Conference on extended Databases*, pages 602–605, 2017.

[147] Bernhard Thalheim. *Entity-Relationship Modeling: Foundations of Database Technology*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2000.

[148] C. A. R. Hoare. Notes on an Approach to Category Theory for Computer Scientists. In *Constructive Methods in Computing Science*, pages 245–305, Berlin, Heidelberg, 1989. Springer.

[149] Neo4j Inc. Cypher Query Language, 2021.

[150] Inc. MongoDB. MongoDB Manual – Query Documents, 2017.

[151] Jean-Marc Hick and Jean-Luc Hainaut. Strategy for Database Application Evolution: The DB-MAIN Approach. In *Conceptual Modeling - ER 2003*, volume 2813 of *Lecture Notes in Computer Science*, pages 291–306, Berlin, Heidelberg, 2003. Springer.

[152] Paolo Atzeni, Giorgio Gianforme, and Paolo Cappellari. A Universal Metamodel and Its Dictionary. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems I*, volume 1, pages 38–62, Berlin, Heidelberg, 2009. Springer.

[153] David Kensche, Christoph Quix, Mohamed Amine Chatti, and Matthias Jarke. GeRoMe: A Generic Role Based Metamodel for Model Management. *Journal on Data Semantics VIII*, 8:82–117, 2007.

[154] World Wide Web Consortium. SPARQL Query Language for RDF, 2008.

[155] ECMA-404 The JSON Data Interchange Standard, 2018. [Online; Accessed 2-February-2018].

[156] Michal Klempa, Michal Kozak, Mário Mikula, Robert Smetana, Jakub Starka, Michal Švirec, Matej Vitásek, Martin Nečaský, and Irena Holubova (Mlýnková). jInfer: A Framework for XML Schema Inference. *The Computer Journal*, 58(1):134–156, 2013.

[157] Kenza Kellou-Menouer, Nikolaos Kardoulakis, Georgia Troullinou, Zoubida Kedad, Dimitris Plexousakis, and Haridimos Kondylakis. A survey on semantic schema discovery. *The VLDB Journal*, 2021.

[158] J. Berstel and L. Boasson. XML Grammars. In *Mathematical Foundations of Computer Science*, Lncs, pages 182–191. Springer, 2000.

[159] Iso. ISO/IEC 9075-1:2008 Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework), 2008.

[160] Dan Brickley and R. V. Guha. RDF Schema 1.1, 2014.

[161] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language Primer (Second Edition), 2012.

[162] Uta Störl, Daniel Müller, Meike Klettke, and Stefanie Scherzinger. Enabling Efficient Agile Software Development of NoSQL-backed Applications. In *Proc. BTW'17*, 2017. Demo.

[163] Alberto Hernández Chillón, Diego Sevilla Ruiz, and Jesús García Molina. Towards a Taxonomy of Schema Changes for NoSQL Databases: The Orion Language. In *Proc. of ER '21, Virtual Event*, volume 13011 of *Lncs*, pages 176–185. Springer, 2021.

[164] Irena Holubová, Pavel Koupil, and Jiaheng Lu. Self-Adapting Design and Maintenance of Multi-Model Databases, 2022.

[165] Brendan Fong and David I Spivak. Seven Sketches in Compositionality: An Invitation to Applied Category Theory, 2018.

List of Figures

0.1 An example of variety of data . . . 10
0.2 An example of ER schema . . . 16
0.3 An example of UML schema . . . 18
0.4 An example of a conceptual schema (Lippe and Ter Hofstede) . . 20
0.5 An example of CGOOD application to multi-model data 23
0.6 An example of a logical categorical schema and data (Spivak et al.) 25
0.7 An example of APG (a) corresponding to data (b) . . . 27
0.8 An example of (a) ETF and (b) EAO strategies . . . 28
0.9 An example of Associative Arrays . . . 30
0.10 An example of TDM . . . 31
0.11 An example of U-Schema for graph (a) and document model (b) . 33
0.12 An example of a single addition (i.e., a pushout) . . . 53
0.13 An example of schema mapping functor (a) inducing pullback (b), right pushforward (c) and left pushforward (d), i.e., data migration functors . . . 55

1.1 Sample multi-model scenario . . . 68
1.2 ER schema diagram for the sample data . . . 68
1.3 Schema category for the sample data . . . 70
1.4 Decomposition into database components . . . 71
1.5 Part of an instance category . . . 72
1.6 Pattern category for a sample query . . . 73
1.7 Query category decomposition . . . 74
1.8 Intermediate and final query results . . . 75
1.9 Alternative query evaluation plan . . . 76
1.10 Old and new versions of schema category . . . 76
1.11 Layered architecture of database systems . . . 77

2.1 A multi-model scenario (partially borrowed from [11]) . . . 80
2.2 Translation of entity types and attributes . . . 87
2.3 Translation of ISA hierarchies . . . 87
2.4 Translation of weak entity types . . . 89
2.5 Translation of relationship types . . . 90
2.6 Translation of the whole ER schema . . . 90
2.7 Architecture of the categorical framework . . . 91

3.1 A sample multi-model scenario . . . 95
3.2 ER schema of the sample multi-model scenario in Figure 3.1 . . . 95
3.3 Schema category extracted from the sample ER schema in Figure 3.2 95
3.4 An example of a category . . . 99
3.5 Grammar of the JSON-like representation of access paths 107
3.6 Collection *Order*, an access path for kind *Order*, and the corresponding part of schema category **S** . . . 108
3.7 Forest containing a single record corresponding to a document . . 109
3.8 Part of instance category **I** after the insertion of single Order . . . 109

3.9 Forest containing three records corresponding to a row of a relational table . . . 111
3.10 Stack M and instance category **I** when the preparation phase for data from Figure 3.7 is completed . . . 115
3.11 Stack M and instance category **I** when the first iteration of tree processing completed . . . 115
3.12 Example of transformation from document model to graph and column models . . . 124
3.13 Sample screen shot of *MM-cat* . . . 125
3.14 Architecture of *MM-cat* . . . 127
3.15 Projection of the result corresponding to the graph pattern Customer − knows → Friend − ordered → Product and its mapping to the graph model . . . 134
3.16 The decomposition of projection pattern and examples of translations of corresponding query parts into particular statements . . . 135
3.17 Joining of intermediate results by pullbacks . . . 136
3.18 The result of the query represented in a graph model . . . 136

4.1 Extended *UniBench* multi-model scenario . . . 143
4.2 An example of relational data . . . 147
4.3 An example of array data . . . 148
4.4 An example of key/value data . . . 148
4.5 An example of column data . . . 148
4.6 An example of document data expressed in the XML format . . . 149
4.7 An example of document data expressed in the JSON format . . . 150
4.8 An example of graph data . . . 150
4.9 An example of multi-model data . . . 151
4.10 An example of Type Hierarchy . . . 155
4.11 An example of RSDs . . . 158
4.12 An example of selected footprints and building of candidates . . . 172
4.13 Example of redundancy, k=1, k=2, k=3 . . . 172
4.14 Screenshots of *MM-infer* . . . 175
4.15 Architecture of *MM-infer* . . . 176
4.16 Results of experiments . . . 184

5.1 An example of multi-model data . . . 188
5.2 ER (a) and categorical (b) schema of sample data . . . 192
5.3 Relational-to-conceptual mapping . . . 193
5.4 Architecture of *MM-evocat* . . . 194
5.5 A screenshot of *MM-evocat* . . . 194
5.6 EBNF syntax of MMSEL . . . 195
5.7 Heavy operation *ungroup Address* . . . 196
5.8 Light operation *group Name, Surname to Personal* . . . 196

A.1 An example of a category . . . 231
A.2 An example of (not) a category . . . 232
A.3 An example of a functor . . . 233
A.4 An example of a set-valued functor (a) and a corresponding graph (b) 234
A.5 An example of commutative triangle (a) and square (b) . . . 234

A.6 Naturality squares for *src* (a) and *tgt* (b) 236
A.7 An example of a natural transformation (a) and the corresponding graph homomorphism (b) 237
A.8 An example of a product corresponding to Cartesian product of sets 240
A.9 An example of a coproduct corresponding to disjoint union of sets 241
A.10 An example of a pullback corresponding to an intersection of sets 242
A.11 An example of a pushout corresponding to union of sets 242

List of Tables

0.1 Unification of terms in popular models 11
0.2 Expressive power of approaches modelling conceptual layer 21
0.3 Comparison of logical layer modeling approaches 34
0.4 Comparison of the selected schema inference approaches 48
0.5 A comparison of features in the selected evolution management approaches 59
0.6 A comparison and classification of supported SMOs in the selected evolution management approaches 59

3.1 Unification of terms in popular models 98
3.2 Allowed complexity of mapping in *MongoDB* and *PostgreSQL* . . 128

4.1 Unification of terms in popular models 152
4.2 An example of *PostgreSQL* (relational model) schema mapping . . 177
4.3 An example of *SciDB* (array model) schema mapping 178
4.4 An example of *Neo4j* (graph model) schema mapping 178
4.5 An example of *Redis* (key/value model) schema mapping 179
4.6 An example of *MarkLogic* (XML document model) schema mapping 180
4.7 An example of *MongoDB* (JSON document) schema mapping . . 180
4.8 An example of *Cassandra* (columnar model) schema mapping . . 181
4.9 Statistics of the used data sets 182

5.1 Unification of terms in popular models 192

A.1 Application of basic definitions 232
A.2 Application of functors 235
A.3 Application of natural transformation 237
A.4 Application of universal constructions 242

List of Algorithms

3.1 Model-to-Category Transformation 113
3.2 Function *children()* 116
3.3 Function *traverseAccessPath()* 116
3.4 DDL Algorithm 118
3.5 DML Algorithm 120
3.6 Function *buildStatement()* 121
3.7 IC Algorithm 122

4.1 Record-Based Local Inference Algorithm 161
4.2 Function *merge()* 162
4.3 Property-Based Local Inference Algorithm 164
4.4 Function *aggregateByHierarchicalName()* 165
4.5 Function *addToForest()* 165
4.6 PDF Miner Algorithm 167
4.7 Function *aggregateByHierarchicalName()* 168
4.8 Candidate Builder Algorithm 169
4.9 Function *formsSubset()* 170

List of Abbreviations

AA Associative Arrays. 21, 34, 35

AI Artificial Intelligence. 12, 189, 197

APG Algebraic Property Graph. 24–27, 34, 215, 232, 235, 237, 242

ASCII American Standard Code for Information Interchange. 73

BFS Breadth-First Search. 90, 154

BSON Binary JSON. 43–45, 47, 146

CGOOD Categorical Graph-Oriented Object Data Model. 22, 23, 34, 215, 232, 235, 237, 242

CPER Categorical Path Equivalence Relation. 23

CPU Central Processing Unit. 183

CQL Categorical Query Language. 23, 25, 34, 55

CSV Comma-Separated Values. 67, 82, 125, 138

DBMS Database Management System. v, 7, 21, 52, 58, 61, 62, 64, 66, 67, 69, 71, 80–82, 90–92, 94–96, 102–105, 121–131, 137–139, 142, 151, 153, 157, 158, 174, 176, 177, 188, 189, 191–193, 199

DDL Data Definition Language. 117–119, 125, 128, 130, 147

DFS Depth-First Search. 111, 115, 117, 132

DML Data Manipulation Language. 64, 117, 119, 130

DSL Domain Specific Language. 63

DTD Document Type Definition. 42, 50, 149

EAO Input per Each Aggregated Object. 27, 28, 215

EBNF Extended Backus–Naur Form. 193, 195, 216

ER Entity-Relationship. 14–17, 20, 35, 36, 39, 40, 48, 50, 68–70, 77, 80–83, 85, 89, 90, 92, 94–96, 100–102, 105, 124, 125, 137, 138, 143, 147, 148, 150–152, 173, 191, 192, 199, 215, 216

ETF Input per Each Top-Level Property. 27, 28, 215

FDM Functional Data Model. 14

FORM Fact-Oriented Modelling. 14

GOOD Graph-Oriented Object Database Model. 22

IC Integrity Constraint. 48, 117, 143, 156, 158–160, 166, 189

ICMO Integrity Constraints Modification Operation. 61

IDEF1X Integration DEFinition for Information Modeling. 16

IFO IFO Model. 14

IMDB Internet Movie Database. 181

IRI Internationalized Resource Identifier. 11, 98, 152

JDBC Java Database Connectivity. 126

JSON JavaScript Object Notation. 9, 11, 24, 27, 37, 41–50, 55, 60, 71, 80, 104, 105, 107, 123, 125, 126, 137, 142–146, 149–153, 156, 157, 163, 171, 173, 174, 176, 177, 179–181, 190, 192, 193, 195–197, 215, 216, 219

JVM Java Virtual Machine. 183

MDE Model-Driven Engineering. 43, 48, 145, 146

MMSEL Multi-Model Schema Evolution Language. 57, 63, 193, 195, 216

NIAM Natural language Information Analysis Method. 14

NoAM NoSQL Abstract Model. 21, 27, 29, 33–35, 71, 81

NoSQL Not only SQL. 7, 9, 21, 27, 41, 43, 46, 48, 52, 57, 61, 63, 68, 71, 76, 80, 81, 94, 137, 139, 144–146, 185, 190

OCL Object Constraint Language. 20, 36, 47, 185

OWL Web Ontology Language. 151

PBA Property-Based Algorithm. 51, 163, 181, 185

PDF Property Domain Footprint. 51, 166, 167, 177

PIM Platform-Independent Model. 14, 190

PSM Platform-Specific Model. 20, 190

RAM Random-Access Memory. 183

RBA Record-Based Algorithm. 51, 160, 181, 183, 185

RDBMS Relational Database Management System. 7

RDF Resource Description Framework. 9, 11, 23, 24, 27, 34, 39, 41, 42, 55, 59, 67, 82, 97, 98, 138, 144, 146, 150–152

RDFS RDF Schema. 151

REST Representational State Transfer. 190

RSD Record Schema Description. 51, 156–161, 163, 164, 170, 171, 173, 174, 176–179, 216

SDM Semantic Data Model. 14

SEL Schema Evolution Language. 56

SMO Schema Modification Operation. 52–54, 56–64, 76, 189, 193, 195, 197, 199, 219, 235

SPARQL SPARQL Protocol and RDF Query Language. 92, 138, 200

SQL Structured Query Language. 9, 134, 147

TDM Tensor Data Model. 21, 30, 31, 34, 71, 81, 215

TSV Tab-Separated Values. 181

UML Unified Modeling Language. 14–18, 20, 35, 36, 40, 41, 45, 46, 48, 50, 68, 70, 77, 81, 92, 94–96, 143, 152, 173, 215

UMP Universal Mapping Property. 240

UP Universal Property. 240

URI Uniform Resource Identifier. 151

XML eXtensible Markup Language. 9, 11, 21, 24, 27, 37, 41, 42, 48–50, 106, 137, 142–145, 149, 152, 153, 156, 157, 171, 173, 174, 176, 177, 179, 180, 190, 216, 219

List of Publications

Journal Paper I: Pavel Koupil, and Irena Holubová. A unified representation and transformation of multi-model data using category theory. *J Big Data* 9, 61 (2022). doi: 10.1186/s40537-022-00613-3 **(Q1, IF: 14.57, SJR: 2.592)**

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

Journal Paper II: Pavel Koupil, Sebastián Hricko, and Irena Holubová. A Universal Approach for Multi-Model Schema Inference. *J Big Data* (accepted). doi: 10.1186/s40537-022-00645-9 **(Q1, IF: 14.57, SJR: 2.592)**

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

Conference Paper I: Pavel Čontoš (Koupil), and Martin Svoboda. JSON schema inference approaches. *1st International Workshop on Conceptual Modeling for NoSQL Data Stores, CoMoNoS 2020.* Vienna, Austria, November 2020. doi: 10.1007/978-3-030-65847-2_16 **(Workshop@CORE A)**

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

Conference Paper II: Pavel Čontoš (Koupil). Abstract Model for Multi-model Data. *26th International Conference on Database Systems for Advanced Applications, DASFAA 2021.* Taipei, Taiwan, April 2021. doi: 10.1007/978-3-030-73200-4_53 **(PhD Consortium@CORE B)**

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

Conference Paper III: Martin Svoboda, Pavel Čontoš (Koupil), and Irena Holubová. Categorical Modeling of Multi-Model Data: One Model to Rule Them All. *10th International Conference on Model and Data Engineering, MEDI 2021.* Tallinn, Estonia, June 2021. doi: 10.1007/978-3-030-78428-7_15 **(CORE C)**

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

Conference Paper IV: Irena Holubová, Pavel Čontoš (Koupil), and Martin Svoboda. Categorical Management of Multi-Model Data. *25th International Database Engineering & Applications Symposium, IDEAS 2021.* Montreal, Canada, July 2021. doi: 10.1145/3472163.3472166 **(CORE B)**

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

Conference Paper V: Irena Holubová, Pavel Čontoš (Koupil), and Martin Svoboda. Multi-Model Data Modeling and Representation: State of the Art and Research Challenges. *25th International Database Engineering & Applications Symposium, IDEAS 2021.* Montreal, Canada, July 2021. doi: 10.1145/3472163.3472267 **(CORE B)**

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

Conference Paper VI: Pavel Koupil, Martin Svoboda, and Irena Holubová. MM-cat: A Tool for Modeling and Transformation of Multi-Model Data using Category Theory. *24th International Conference on Model Driven Engineering Languages and Systems, MODELS 2021.* Fukuoka, Japan, October 2021. doi: 10.1109/MODELS-C53483.2021.00098 **(CORE A)**

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

Conference Paper VII: Pavel Koupil, and Irena Holubová. Unifying Categorical Representation of Multi-Model Data. *37th ACM/SIGAPP Symposium On Applied Computing, SAC 2022.* Brno, Czech Republic, April 2022. doi: 10.1145/3477314.3507690 **(CORE B)**

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

Conference Paper VIII: Ivan Veinhardt Latták, and Pavel Koupil. A Comparative Analysis of JSON Schema Inference Algorithms. *17th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2022.* Virtual Event, April 2022. doi: 10.5220/0011046000003176 **(CORE B)**

Contribution: The paper is based on Ivan Veinhardt Latták's Master thesis [107] (supervised by Pavel Koupil).

Conference Paper IX: Pavel Koupil, Sebastián Hricko, and Irena Holubová. MM-infer: A Tool for Inference of Multi-Model Schemas. *29th International Conference on Extending Database Technology, EDBT 2022.* Edinburgh, UK, March 2022. doi: 10.48786/edbt.2022.52 **(CORE A)**

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

Conference Paper X: Irena Holubová, Pavel Koupil, and Jiaheng Lu. Self-Adapting Design and Maintenance of Multi-Model Databases. *26th International Database Engineering & Applications Symposium, IDEAS 2022.* (accepted) **(CORE B)**

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

Conference Paper XI: Pavel Koupil, Sebastián Hricko, and Irena Holubová. Schema Inference for Multi-Model Data. *25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022.* (accepted) **(CORE A)**

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

Conference Paper XII: Pavel Koupil, Jáchym Bártík, and Irena Holubová. *MM-evocat*: A Tool for Modelling and Evolution Management of Multi-Model Data. Manuscript under review.

Contribution: The share of the author's contributions in this paper is equal.

A. Category Theory

Category theory [17] is a branch of mathematics that provides a way to generalise mathematical structures and the relationships between them. Hence, it is a unifying theory that is useful for finding connections between different areas, not only in mathematics and theoretical computer science. We assume that by applying category theory we will achieve a reasonable level of abstraction of various data models and their combinations that will allow us to perform data migration, querying, and evolution management of multi-model data in a unified way.

In this section, we provide the basic definitions underlying our approach of multi-model schema and data representation (see Chapter 2), data migration ★ (see Chapter 3), and schema and data evolution (see Chapter 5), as well as the ★ approaches we have been inspired by [18, 19, 20, 21]. **Note that our aim is to propose an intuitive and user-friendly approach. Therefore, we explicitly use only the most basic constructs of category theory, i.e., primarily categories** (see Definition 1) **and functors** (see Definition 6). As for the other constructs, i.e., natural transformations (see Definition 10) and universal constructions (see Definitions 13, 14, 16, and 17), these are considered only implicitly, i.e., we do not require the user to have active knowledge of these definitions. On the contrary, these complex constructs are explicitly utilised in the approaches we are inspired by.

In addition, we also provide illustrative examples that are closely related to real-world applications in data modelling approaches at the conceptual and logical level. Finally, for the convenience of the reader, at the end of each subsection we outline in which approaches and for which purpose the above definitions are applied, i.e., we provide additional examples.

For more details we refer an interested reader to [165, 124, 123, 133, 17], which are ordered by difficulty, while for computer science we recommend particularly [165].

A.1 Basic Definitions

In this subsection, we introduce the basic definitions and concepts which form the foundation of category theory. Specifically, we introduce the notion of a category as a collection of objects and morphisms (sometimes called arrows), and we introduce different classes of morphisms based on their properties, as well as a way to create a category from an arbitrary graph. We conclude with illustrative examples of categories.

Definition 1. *The **category** $\mathbf{C}$ is a quadruple $(\mathcal{O}_\mathbf{C}, \mathcal{M}_\mathbf{C}, \circ, 1)$ such that:*

- *$\mathcal{O}_\mathbf{C}$ is a collection of **objects**.*
- *$\mathcal{M}_\mathbf{C}$ is a collection of **morphisms** where each $f \in \mathcal{M}_\mathbf{C}$ is represented as an arrow $f : A \to B$ (also denoted $A \xrightarrow{f} B$), where $A, B \in \mathcal{O}_\mathbf{C}$, such that A is the domain of f, denoted by $dom(f) = A$, and B is the codomain of f, denoted by $cod(f) = B$.*

- *Given* $f, g \in \mathcal{M}_{\mathbf{C}}$ *and* $cod(f) = dom(g)$*, there exists* $g \circ f \in \mathcal{M}_{\mathbf{C}}$*, which we refer to as the* ***composition*** *of* f *and* g*. Moreover, the composition must be associative, i.e., for any* $f, g, h \in \mathcal{M}_{\mathbf{C}}$ *such that* $cod(f) = dom(g)$ *and* $cod(g) = dom(h)$*, the equality* $h \circ (g \circ f) = (h \circ g) \circ f$ *holds.*

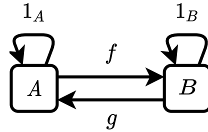
- *For each object* $A \in \mathcal{O}_{\mathbf{C}}$*, there is exactly one* ***identity*** *morphism* $1_A : A \to A$ *such that* $f \circ 1_A = f = 1_B \circ f$ *holds for any* $f : A \to B$*,* $f \in \mathcal{M}_{\mathbf{C}}$*.*

Definition 2. *Let* $\mathbf{C}$ *be a category and* $A, B \in \mathcal{O}_{\mathbf{C}}$*. Then we define the* ***hom-class*** $hom_{\mathbf{C}}(A, B) \subset \mathcal{M}_{\mathbf{C}}$ *as the collection of all morphisms* $f : A \to B$*.*

Definition 3. *We call the category* $\mathbf{C}$ *a* ***small category*** *if both* $\mathcal{O}_{\mathbf{C}}$ *and* $\mathcal{M}_{\mathbf{C}}$ *are sets. Otherwise, we call the category* $\mathbf{C}$ *a* ***large category****. We say that* $\mathbf{C}$ *is* ***locally small category*** *if for any two objects* $A, B \in \mathcal{O}_{\mathbf{C}}$ *it holds that* $hom_{\mathbf{C}}(A, B)$ *forms a set.*

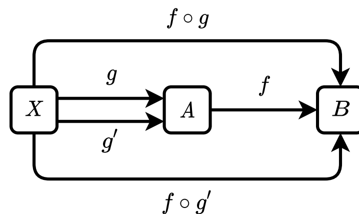
Definition 4. *Let* $\mathbf{C}$ *be a category,* $A, B, X \in \mathcal{O}_{\mathbf{C}}$ *and* $f, g, g', g'' \in \mathcal{M}_{\mathbf{C}}$*.*

A morphism $f : A \to B$ *is an* ***isomorphism*** *if and only if there exists a morphism* $g : B \to A$ *such that the composition of* f *and* g *yields identities, i.e.,* $g \circ f = 1_A$ *and* $f \circ g = 1_B$*. Moreover, the morphism* g *is uniquely determined. That is, if there exist* $g', g'' : B \to A$ *such that* $g' \circ f = 1_A$ *and* $f \circ g'' = 1_B$*, then it must hold that* $g' = g' \circ 1_B = g' \circ f \circ g'' = 1_A \circ g'' = g''$*. We will denote the morphism* g *by* f^{-1} *and call it the* ***inverse morphism*** *of* f*.*



If there is a pair of isomorphisms $f : A \to B$*,* $g : B \to A$*, then also* A *is isomorphic to* B*, which we denote by* $A \cong B$*. Note also that identity morphisms are trivial isomorphisms.*

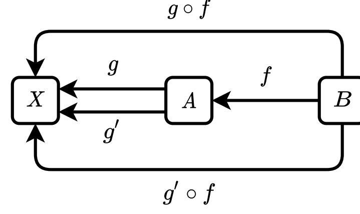
A morphism $f : A \to B$ *is a* ***mono****(morphism),*[1] *if for any object* X *and arbitrary two morphisms* $g, g' : X \to A$ *the following implication holds:* $f \circ g = f \circ g' \implies g = g'$*.*



A morphism $f : B \to A$ *is an* ***epi****(morphism),*[2] *if for any object* X *and arbitrary pair of morphisms* $g, g' : A \to X$ *the following implication holds:* $g \circ f = g' \circ f \implies g = g'$*.*

[1] A special case of a monomorphism is an injective morphism, but not every morphism is an injective morphism. Monomorphism is a more general notion than injection.

[2] A special case of epimorphism is a surjective morphism, but not every epimorphism is a surjective morphism. An epimorphism is a more general notion than surjection.



Definition 5. *Let $G = (V, E, src, tgt)$ be a graph such that V is the set of vertices, E is the set of edges, and $src : E \to V$, $tgt : E \to V$ are functions assigning the source vertex and the target vertex to an edge.*

*The category $\mathbf{Free(G)} = \{\mathcal{O}_{\mathbf{Free(G)}}, \mathcal{M}_{\mathbf{Free(G)}}, \circ, 1\}$, referred to as the **free category** on G, is the category with $\mathcal{O}_{\mathbf{Free(G)}}$ equal to V, $hom_{\mathbf{Free(G)}}(A, B)$, equal to all paths from v_a to v_b in G, such that $A, B \in \mathcal{O}_{\mathbf{Free(G)}}$ and $v_a, v_b \in V$, composition is determined by the concatenation of paths, and identity morphisms $1_A : A \to A$, $1_A \in \mathcal{M}_{\mathbf{Free(G)}}$ on an object is the trivial path at $v_a \in V$.*

Example A.1. The category **Set** is a category in which objects are sets and morphisms are functions between the sets. The composition of morphisms is given by the composition of functions, and the identity morphism is the identity function. Note also that the category **Set** has both initial and terminal objects. □

Example A.2. Figure A.1 illustrates a category $\mathbf{G} = (\mathcal{O}_\mathbf{G}, \mathcal{M}_\mathbf{G}, \circ_\mathbf{G}, 1_\mathbf{G})$, where $E, V \in \mathcal{O}_\mathbf{G}$ and $1_E, 1_V, src, tgt \in \mathcal{M}_\mathbf{G}$. The category $\mathbf{G}$ is indeed a category as the identity morphisms over all vertices are defined, i.e., 1_E and 1_V, and the associativity law for morphism composition holds.

Also note that the category $\mathbf{G}$ corresponds to the schema of an arbitrary directed graph $G = (V_G, E_G, src_G, tgt_G)$, i.e., the objects V and E correspond to the elements of the graph V_G and E_G and the morphisms src and tgt represent the functions src_G and tgt_G. We will show a particular graph in Example A.5 after we define the notion of a functor between categories. □

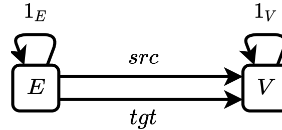


Figure A.1: An example of a category

Example A.3. Although a category is a special type of directed multi-graph, not every directed (multi-)graph is a category. Let the graph $G = (V, E, src, tgt)$ consist of vertices $A, B, C \in V$ and edges $f, g \in E$ (see Figure A.2 (a)). In this instance, it is not a category, e.g., because the graph does not contain a reflexive edge at any vertex that would correspond to an identity morphism.

Figure A.2 (b) illustrates the graph G', where $E' = E \cup \{1_A, 1_B, 1_C\}$. Again, this is not a category, as the path formed by the composition of the paths of the graph G', e.g., $g \circ f$, is not included in the graph.

Finally, Figure A.2 (c) illustrates the graph G'', where $E'' = E' \cup \{h\}$, $h = g \circ f$, which corresponds to the category. In other words, the graph G freely generates the category $\mathbf{Free(G)}$ (see Definition 5). □

Finally, Table A.1 summarises the application of categories in approaches representing data at the conceptual or logical level.

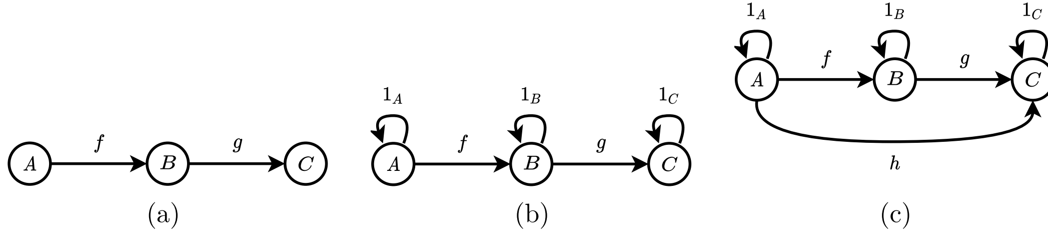


Figure A.2: An example of (not) a category

Table A.1: Application of basic definitions

	Lipppe and Ter Hofstede [18]	**CGOOD** [19]	**Spivak et al.** [20]	**APG** [21]	**MM-(evo)cat** [3, 5]
Category (Definition 1)	Conceptual schema	Abstraction of schema	Schema	Abstraction of property labelled graph	Schema and instance category (early approach)
Small category (Definition 3)	-	-	Schema	-	Schema and instance category
Monomorphism (Definition 4)	Uniqueness	-	-	-	Uniqueness, inheritance
Epimorphism (Definition 4)	Simple and structured attributes	-	-	-	Simple and structured attributes
Free category (Definition 5)	-	-	Schema	-	Schema category

A.2 Functors

So far we have defined the elementary constructs of category theory. In this subsection, we add functors, i.e., structure-preserving mappings between categories that (not coincidentally) resemble morphisms between objects. Once again, we conclude with illustrative examples.

Definition 6. *Let* $\mathbf{C} = \{\mathcal{O}_\mathbf{C}, \mathcal{M}_\mathbf{C}, \circ, 1\}$ *and* $\mathbf{D} = \{\mathcal{O}_\mathbf{D}, \mathcal{M}_\mathbf{D}, \circ, 1\}$ *be categories. A* ***functor*** $F : \mathbf{C} \to \mathbf{D}$*, also denoted* $\mathbf{C} \xrightarrow{F} \mathbf{D}$*, is a structure-preserving mapping between categories that assigns objects* $\mathcal{O}_\mathbf{C}$ *to objects in* $\mathcal{O}_\mathbf{D}$ *and morphisms in* $\mathcal{M}_\mathbf{C}$ *to morphisms in* $\mathcal{M}_\mathbf{D}$*. For the functor* F*, the following holds:*

- $dom(F(f)) = F(dom(f))$ *and* $cod(F(f)) = F(cod(f))$ *for each morphism* $f \in \mathcal{M}_\mathbf{C}$*.*
- *Preserving of composition* $F(g \circ f) = F(g) \circ F(f)$ *for every pair of morphisms* $f, g \in \mathcal{M}_\mathbf{C}$ *such that* $dom(g) = cod(f)$*.*
- *Preserving of identity* $F(1_A) = 1_{F(A)}$ *for each object* $A \in \mathcal{O}_\mathbf{C}$*.*

Definition 7. *Let* $\mathbf{C}$ *and* $\mathbf{D}$ *be categories and* $F : \mathbf{C} \to \mathbf{D}$ *and* $G : \mathbf{D} \to \mathbf{E}$ *be functors. The* ***composition of the functors*** F *and* G *is a functor* $G \circ F : \mathbf{C} \to \mathbf{E}$ *such that:*

- *For every object $A \in \mathcal{O}_{\mathbf{C}}$ it holds that $G \circ F(A) = G(F(A))$.*
- *For every morphism $f : A \to B$, $f \in \mathcal{M}_{\mathbf{C}}$ it holds that $G \circ F(f) = G(F(f))$.*

Definition 8. *Let $\mathbf{C}$ and $\mathbf{J}$ be categories. The diagram of the form $\mathbf{J}$ in the category $\mathbf{C}$ is a functor $D : \mathbf{J} \to \mathbf{C}$. We refer to $\mathbf{J}$ as the index category of the diagram D.*

Definition 9. *We say that a* ***diagram commutes*** *if every two paths $p = f_m \circ ... \circ f_1$ and $q = g_n \circ ... \circ g_1$, where $m, n \in \mathbb{N}$, $f_1, \dots, f_m, g_1, \dots, g_m, p, q \in \mathcal{M}_{\mathbf{C}}$, $dom(p) = dom(q)$, and $cod(p) = cod(q)$, determine the same morphism via composition, i.e., $p = q$.*

Example A.4. Figure A.3 illustrates examples of functors $F : \mathbf{C} \to \mathbf{D}$ (depicted in blue). Figure A.3 (a) corresponds to usual expectation of how functor $F : \mathbf{C} \to \mathbf{D}$ should map objects between two categories. Figure A.3 (b) illustrates functor F mapping objects A, B to B' and the morphism f to the identity morphism $1_{B'}$. Figure A.3 (c) maps object A to A' and B to C' with morphism f being mapped to the composition of morphisms $g' \circ f'$. In contrast, Figure A.3 (d) does not represent a functor because (1) object A is mapped to more than one object and (2) object B is not mapped at all. Figure A.3 (e) does not illustrate the functor either, since the structure of the category $\mathbf{C}$ is not preserved, i.e., the morphism f cannot be mapped. □

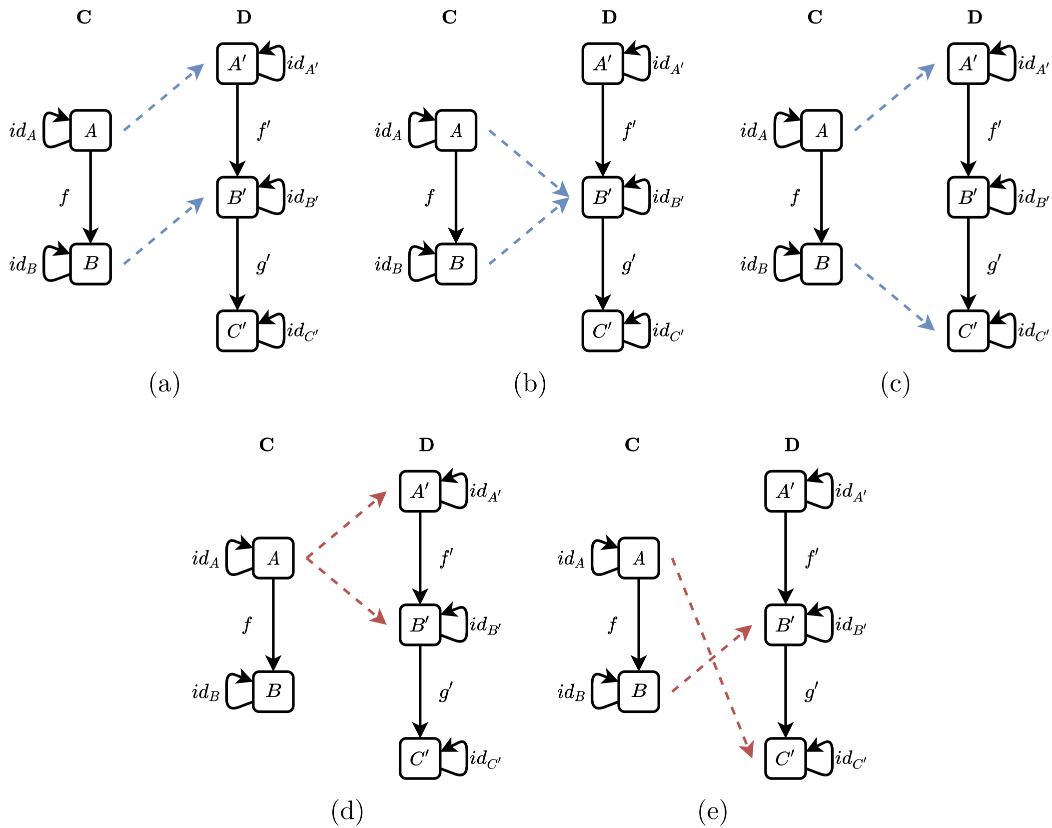


Figure A.3: An example of a functor

Example A.5. An example of a functor is the *set-valued functor* $Inst : \mathbf{G} \rightarrow \mathbf{Set}$, where $\mathbf{G}$ is the category from Example A.2 and $\mathbf{Set}$ is the category of all sets from Example A.1. It maps each object of category $\mathbf{G}$ to an object of category $\mathbf{Set}$, i.e., it assigns the object E to the set (of edges) $Inst(E) = E_{\mathbf{Set}}$ and the object V to the set (of vertices) $Inst(V) = V_{\mathbf{Set}}$ (see Figure A.4 (a)). The corresponding graph, represented by the (one of many) functor $Inst : \mathbf{G} \rightarrow \mathbf{Set}$, is depicted in Figure A.4 (b). (Note that for clarity only, we do not depict identity morphisms in the figure.) □

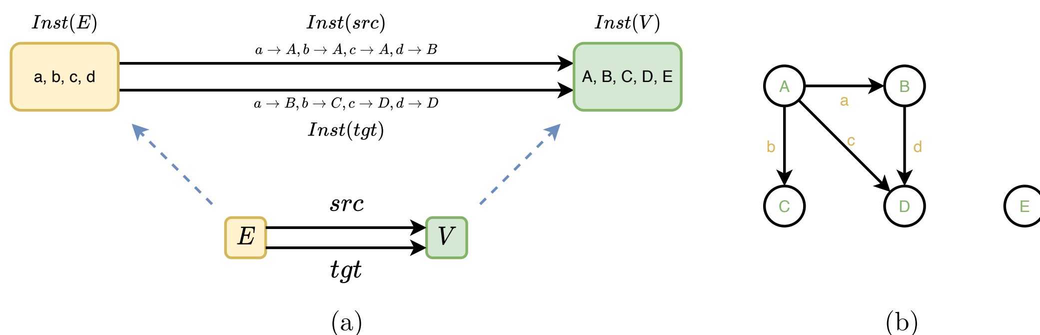


Figure A.4: An example of a set-valued functor (a) and a corresponding graph (b)

Example A.6. The commutativity of the diagram depicted in Figure A.5 (a) implies that $g \circ f = h$, i.e., $dom(g \circ f) = dom(h)$ and $cod(g \circ f) = cod(h)$ and both $g \circ f$ and h lead to the same result. Note that this commutative diagram is referred to as a *commutative triangle*.

Similarly, the commutativity of the diagram depicted in Figure A.5 (b) implies that $g \circ f = f' \circ g'$, where $dom(g \circ f) = dom(f' \circ g')$ and $cod(g \circ f) = cod(f' \circ g')$. The commutative diagram is referred to as a *commutative square*. □

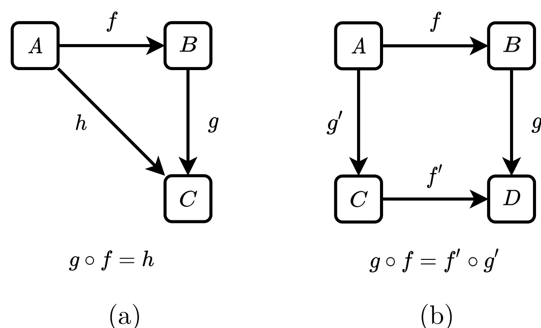


Figure A.5: An example of commutative triangle (a) and square (b)

To conclude, Table A.2 summarises the application of functors in approaches representing data at the conceptual or logical level and in approaches performing data migration.

Table A.2: Application of functors

	Lippe and Ter Hofstede [18]	**CGOOD** [19]	**Spivak et al.** [20]	**APG** [21]	**MM-(evo)cat** [3, 5]
Functor (Definition 6)	-	Particular schema, data, and data instance	Particular data instance; SMOs; data migration	Particular property labelled graph	Particular data instance (evolution approach); SMOs; data migration

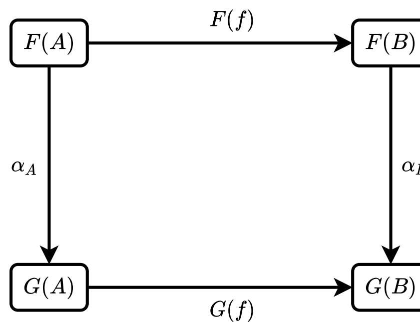
A.3 Natural Transformations

In this subsection, we define natural transformations that allow us to have multiple views of the same concept using different levels of abstraction. In addition, we define vertical composition of natural transformations, natural isomorphism, and functor category, where the objects of this category are functors and the morphisms are natural transformations. We conclude with illustrative examples of universal transformations and a functor category.

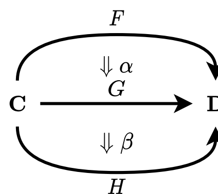
Definition 10. *Let* $\mathbf{C}$ *and* $\mathbf{D}$ *be categories and let* $F : \mathbf{C} \to \mathbf{D}$ *and* $G : \mathbf{C} \to \mathbf{D}$ *be functors.*

The ***natural transformation*** α *from* F *to* G*, denoted* $\alpha : F \to G$*, is a collection of components (i.e., morphisms)* α_A*,* $A \in \mathcal{O}_{\mathbf{C}}$*, that satisfy the commutative square law, as follows:*

- *For each object* $A \in \mathcal{O}_{\mathbf{C}}$ *there is a morphism* $\alpha_A : F(A) \to G(A)$*,* $\alpha_A \in \mathcal{M}_{\mathbf{D}}$*, called the* A***-component*** *of the natural transformation* α*.*
- *For each morphism* $f : A \to B$*,* $f \in \mathcal{M}_{\mathbf{C}}$*,* $A, B \in \mathcal{O}_{\mathbf{C}}$*, the following square, called* ***naturality square****, commutes, i.e.,* $\alpha_B \circ F(f) = G(f) \circ \alpha_A$*.*



Definition 11. *Let* $\mathbf{C}$ *and* $\mathbf{D}$ *be categories, let* $F, G, H : \mathbf{C} \to \mathbf{D}$ *be functors, and let* $\alpha : F \to G$ *and* $\beta : G \to H$ *be natural transformations.*



Vertical composition *of natural transformations* $\beta \circ \alpha$ *is a composition in which for each object* $c \in \mathcal{O}_{\mathbf{C}}$ *there exists a morphism* $(\beta \circ \alpha)_c \in \mathcal{M}_{\mathbf{D}}$ *for which* $(\beta \circ \alpha)_c = \beta_c \circ \alpha_c$ *(i.e., it commutes).*

Definition 12. *Let* $\mathbf{C}$ *and* $\mathbf{D}$ *be categories and let* $F : \mathbf{C} \to \mathbf{D}$ *and* $G : \mathbf{C} \to \mathbf{D}$ *be functors. A natural transformation* $\alpha : F \to G$ *is said to be a* ***natural isomorphism*** *if every component of* $\alpha_A : F(A) \to G(A)$ *is an isomorphism. In that case, the functors* F *and* G *are called* ***naturally isomorphic****.*

Lemma 1. *Let* $\mathbf{C}$ *and* $\mathbf{D}$ *be categories,* $F, G, H : \mathbf{C} \to \mathbf{D}$ *functors,* $\alpha : F \to G$ *and* $\alpha' : G \to H$ *natural transformations, and* $1_F : F \to F$ *a natural isomorphism.*

There exists a ***functor category***[3]*, denoted* $\mathbf{D}^{\mathbf{C}}$*, such that:*

- $\mathcal{O}_{\mathbf{D}^{\mathbf{C}}}$ *is a collection of functors* $F : \mathbf{C} \to \mathbf{D}$*.*
- $\mathcal{M}_{\mathbf{D}^{\mathbf{C}}}$ *is a collection of natural transformations* $\alpha : F \to G$*.*
- *The composition* $\alpha' \circ \alpha$ *is a vertical composition of natural transformations.*
- *The identity* 1_F *on the object* F *is a natural isomorphism.*

Proof. See [123]. □

Example A.7. Let $\mathbf{G}$ be the category from Example A.2, $\mathbf{Set}$ be the category from Example A.1 and let $Inst, Inst' : \mathbf{G} \to \mathbf{Set}$ be functors from Example A.5.

The natural transformation $\alpha : Inst \to Inst'$ involves two components, $\alpha_E : Inst(E) \to Inst'(E)$ and $\alpha_V : Inst(V) \to Inst'(V)$ and two naturality squares, $\alpha_V \circ Inst(src) = Inst'(src) \circ \alpha_E$ (see Figure A.6 (a)) and $\alpha_V \circ Inst(tgt) = Inst'(tgt) \circ \alpha_E$ (see Figure A.6 (b)).

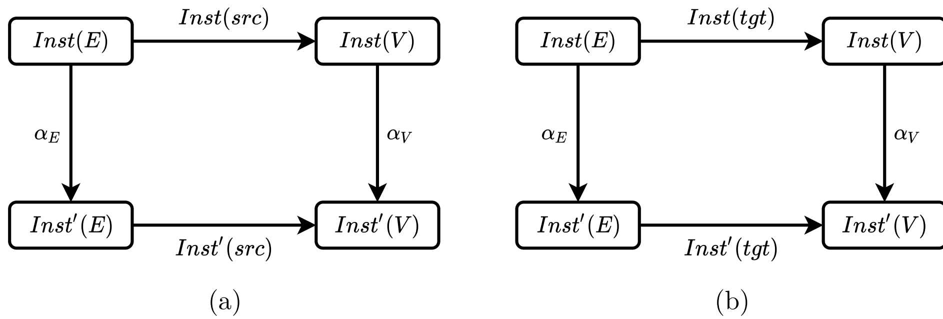


Figure A.6: Naturality squares for src (a) and tgt (b)

In other words, the natural transformation $\alpha : Inst \to Inst'$ (see Figure A.7 (a)) is the same as the graph homomorphism [124] (see Figure A.7 (b)).

□

Example A.8. Let $\mathbf{G}$ be the category from Example A.2, $\mathbf{Set}$ be the category of all sets from Example A.1, let $Inst, Inst', Inst'' : \mathbf{G} \to \mathbf{Set}$ be functors from

[3] Note that there exists also horizontal composition of natural transformations and a category, where the objects are categories and the morphisms are horizontal natural transformations of functors [123].

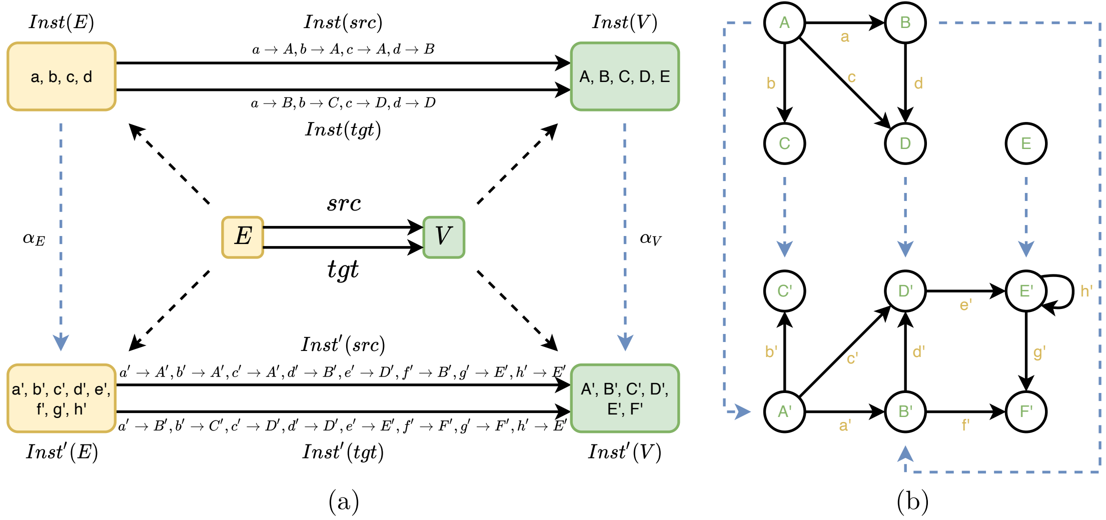


Figure A.7: An example of a natural transformation (a) and the corresponding graph homomorphism (b)

Example A.5 (i.e., particular graphs) and let $\alpha : Inst \rightarrow Inst'$ a $\beta : Inst' \rightarrow Inst''$ be natural transformations from Example A.7 (i.e., graph homomorphisms).

The category of all graphs is the functor category $\mathbf{Set}^{\mathbf{G}}$ such that the objects of this category are all graphs and the morphisms are graph homomorphisms. □

Finally, Table A.3 summarises the application of natural transformation in approaches representing data at the conceptual or logical level and in approaches performing data migration.

Table A.3: Application of natural transformation

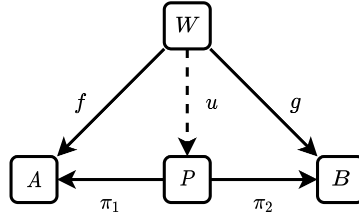
	Lippe and Ter Hofstede [18]	**CGOOD** [19]	**Spivak et al.** [20]	**APG** [21]	**MM-(evo)cat** [3, 5]
Natural transformation (Definition 10)	-	Allowed operations between schemas	Allowed operations between data instances conforming to the same schema	Allowed operations between property labelled graphs	Allowed operations between data instances conforming to the same schema (evolution approach)
Functor category (Lemma 1)	-	Category of all schemas	Category of all instances conforming to the same schema	Category of all property labelled graphs	Category of all instances conforming to the same schema (evolution approach)

A.4 Universal Constructions

In this subsection, we define so-called *universal constructions* that may resemble notions from set theory, such as, e.g., Cartesian product, disjunctive union,

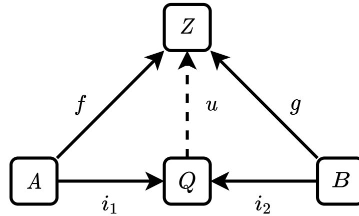
intersection, and union. We then conclude with examples to show that this similarity is not coincidental, but the categorical constructions are abstractions of (not only) these concepts.

Definition 13. *Let* $\mathbf{C}$ *be a category and* $A, B \in \mathcal{O}_{\mathbf{C}}$ *be objects. The* ***product*** *of two objects* A *and* B *consists of an object* P *and morphisms* $\pi_1 : P \to A$, $\pi_2 : P \to B$ *such that for any object* $W \in \mathcal{O}_{\mathbf{C}}$ *with morphisms* $f : W \to A$, $g : W \to B$, $f, g \in \mathcal{M}_{\mathbf{C}}$. *Moreover, there exists a unique morphism* $u : W \to P$, $u \in \mathcal{M}_{\mathbf{C}}$ *such that the diagram commutes, that is, such that* $f = \pi_1 \circ u$ *and* $g = \pi_2 \circ u$.



The object P *is usually denoted by* $A \times B$ *and the morphisms* π_1 *and* π_2 *are referred to as* ***projections***.

Definition 14. *Let* $\mathbf{C}$ *be a category and* $A, B \in \mathcal{O}_{\mathbf{C}}$ *be objects. The* ***coproduct*** *of two objects* A *a* B *consists of an object* Q *and morphisms* $i_1 : A \to Q$, $i_2 : B \to Q$ *such that for any object* $Z \in \mathcal{O}_{\mathbf{C}}$ *with morphisms* $f : A \to Z$, $g : A \to Z$, $f, g \in \mathcal{M}_{\mathbf{C}}$. *Moreover, there exists a unique morphism* $u : Q \to Z$, $u \in \mathcal{M}_{\mathbf{C}}$ *such that the diagram commutes, that is, such that* $f = u \circ i_1$ *and* $g = u \circ i_2$.



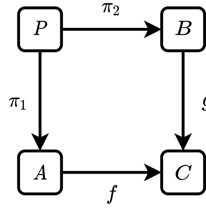
The object Q *is usually denoted by* $A + B$ *and the morphisms* i_1 *and* i_2 *are referred to as* ***injections****, even though they do not need to be injective.*

Remark: In the literature, coproduct is also referred to as a sum [124].

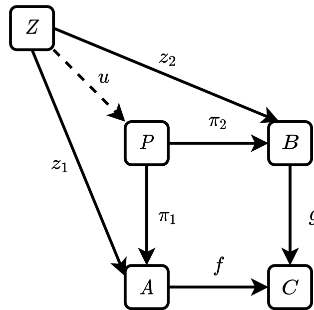
Definition 15. *A morphism* $f : A \to Z$ *is* ***complementable*** *if and only if there exists a morphism* $g : B \to Z$ *such that* $Z \cong A + B$, *where* f *and* g *are injection morphisms. We refer to the morphism* g *as the* ***complement*** *of* f *and the object* B *is often denoted as* $Z - A$.

Definition 16. *Let* $\mathbf{C}$ *be a category, let* $A, B, C \in \mathcal{O}_{\mathbf{C}}$ *be objects, and let* $f : A \to C$ *and* $g : B \to C$, $f, g \in \mathcal{M}_{\mathbf{C}}$ *be morphisms.*

The ***pullback*** *of* $A \xrightarrow{f} C \xleftarrow{g} B$ *is* $A \xleftarrow{\pi_1} P \xrightarrow{\pi_2} B$ *such that* $f \circ \pi_1 = g \circ \pi_2$, *i.e., such that the square commutes.*



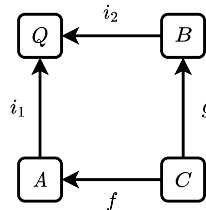
Moreover, it must hold that for any object $Z \in \mathcal{O}_{\mathbf{C}}$ and two morphisms $z_1 : Z \to A$ and $z_2 : Z \to B$, $z_1, z_2 \in \mathcal{M}_{\mathbf{C}}$ such that $f \circ z_1 = g \circ z_2$, there exists a unique morphism $u : Z \to P$ such that $z_1 = \pi_1 \circ u$ and $z_2 = \pi_2 \circ u$.



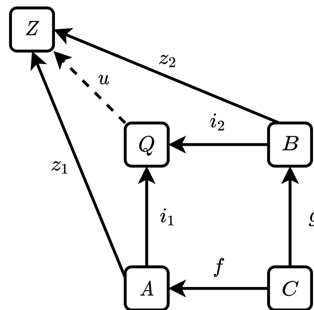
The object P is usually denoted by $A \times_C B$.

Definition 17. *Let $\mathbf{C}$ be a category, let $A, B, C \in \mathcal{O}_{\mathbf{C}}$ be objects, and let $f : C \to A$, $g : C \to B$, $f, g \in \mathcal{M}_{\mathbf{C}}$ be morphisms.*

*The **pushout** of $A \xleftarrow{f} C \xrightarrow{g} B$ is $A \xrightarrow{i_1} Q \xleftarrow{i_2} B$ such that $i_1 \circ f = i_2 \circ g$, that is, such that the square commutes.*



Moreover, it must hold that for any object $Z \in \mathcal{O}_{\mathbf{C}}$ and two morphisms $z_1 : A \to Z$ and $z_2 : A \to Z$, $z_1, z_2 \in \mathcal{M}_{\mathbf{C}}$ such that $z_1 \circ f = z_2 \circ g$, there exists a unique morphism $u : Q \to Z$ such that $z_1 = u \circ i_1$ and $z_2 = u \circ i_2$.



The object Q is usually denoted by $A +_C B$.

Note that universal constructs are characterised by the existence of a unique morphism u. This feature is referred to as the universal property UP or the universal mapping property UMP. Finally, as universal constructs are given by a universal property, they are unique up to the unique isomorphism [123].

Example A.9. Let $C = \{a, b, c\}$ and $R = \{1, 2\}$ be sets. Figure A.8 illustrates an example of the product of sets C and R, i.e., the Cartesian product $C \times R = \{(c, r)|c \in C, r \in R\}$ together with the projections $\pi_1 : C \times R \to C$ (red morphism) a $\pi_2 : C \times R \to R$ (green morphism). Moreover, if there is another candidate product (not necessarily a Cartesian product), e.g., $W = \{k\}$ with projections $f : W \to R$ (yellow morphism) and $g : W \to C$ (blue morphism), then there exists a universal mapping $h : W \to R \times C$ (purple morphism) such that the diagram commutes.

At first sight, the universal property of products may not be intuitive. As an example, let us give a real-world meaning to the sets C, R and W. Imagine that we have a set of game pieces W that we want to place on a board with assigned coordinates $Column = C$ and $Row = R$, i.e., each field can be identified as a pair $(c, r) \in Column \times Row$, where $\pi_1 : Column \times Row \to Column$ returns the column coordinate and $\pi_2 : Column \times Row \to Row$ returns the row coordinate. Placing the game pieces at the coordinates $c \in Column$ and $1 \in Row$, i.e., an application of the functions $f : W \to Column$ and $g : W \to Row$ corresponds to applying function $h : W \to Column \times Row$, i.e., selecting the board field $(c1)$.

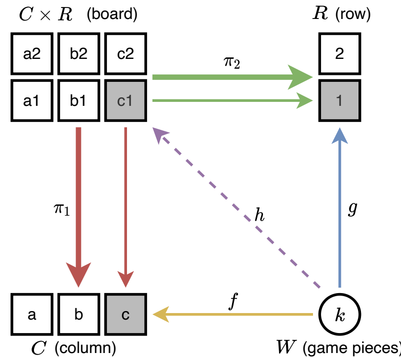


Figure A.8: An example of a product corresponding to Cartesian product of sets

Another example of a product of sets C and R is the Cartesian product $R \times C = \{(r, c)|r \in R, c \in C\}$. It is easy to prove that due to the universal product property, both products are isomorphic, i.e., $R \times C \cong C \times R$.

Finally, there also exists products of more than just two sets (i.e., objects). For example, let A, B, and C be sets. Then, e.g., $A \times B \times C = \{(a, b, c)|a \in A, b \in B, c \in C\}$ is the product of these sets. $\square$

Example A.10. Let $B = \{1, 2, 3, 4\}$ and $W = \{1, 2\}$ be sets. Figure A.9 illustrates an example of the coproduct of sets B and W, i.e., disjunctive union $B + W = (B \times \{\blacksquare\} \cup (W \times \{\square\})$, where $\{\blacksquare, \square\}$ denotes the origin of the element (i.e., $\blacksquare$ is assigned to each $b \in B$ and $\square$ is assigned to each $w \in W$), together with the inclusions $i_1 : B + W \to B$ (red arrows) a $i_2 : B + W \to W$ (green arrows). Moreover, if there is another candidate coproduct (not necessarily a disjunctive union), e.g., $Z = \{r, n, b, k, q\}$ with inclusions $f : B \to Z$ (yellow arrows) and

$g : W \to Z$ (blue arrows), then it holds that there exits a universal mapping $h : B + W \to Z$ (purple arrows) such that the diagram commutes.

Let us again illustrate the coproduct using a real example. Suppose the set B represents the (sub)set of black chess pieces and the set W represents the (sub)set of white chess pieces. The set of all pieces is their disjunctive union $B + W$, i.e., the (sub)set of chess pieces. A candidate coproduct may be the set $Z = \{rock, knight, bishop, king, queen\}$, together with morphisms $f : A \to Z$ and $g : B \to Z$, which determine the type of the pieces, i.e., r is a $rock$, n is a $knight$, b is a $bishop$, k is a $king$, and q is a $queen$. Then there exists a unique morphism $h : B+W \to Z$ which states that the type of a piece can be determined for all chess pieces, i.e., the diagram commutes.

Moreover, similar to Example A.9, there can be multiple coproducts that are isomorphic to each other. It is also possible to construct a coproduct for more than two sets (i.e., objects). □

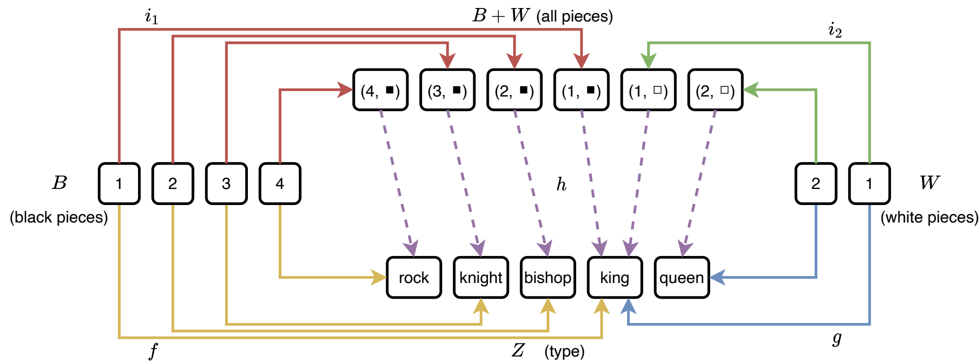


Figure A.9: An example of a coproduct corresponding to disjoint union of sets

Example A.11. Let $A = \{1, 2, 3, 4\}$ and $B = \{3, 4, 5, 6\}$ be sets and let $C = \{1, 2, 3, 4, 5, 6\}$ be their union, i.e., there exist inclusive mappings $f : A \to C$ and $g : B \to C$.

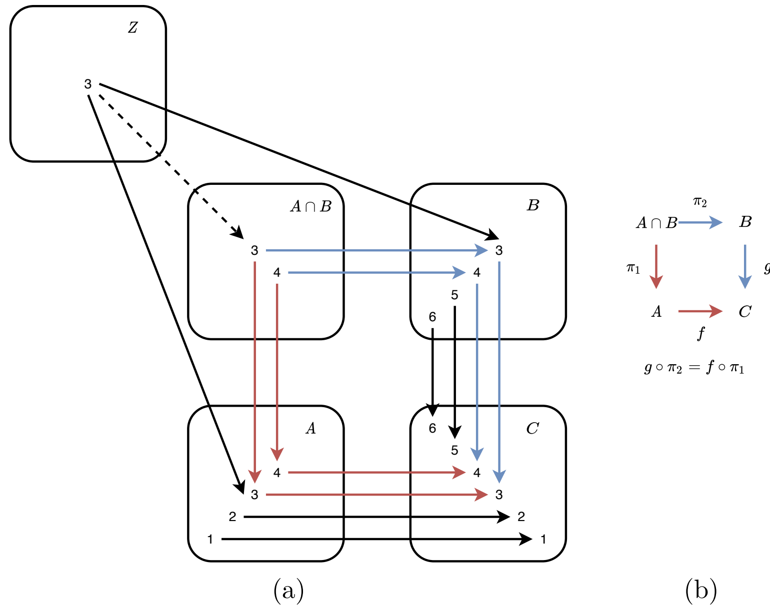
The pullback of $A \xleftarrow{f} C \xrightarrow{g} B$ is, e.g., $A \xrightarrow{\pi_1} P \xleftarrow{\pi_2} B$, where P consists of elements p such that $f(a) = g(b) = p$, i.e., $P = \{3, 4\} = A \cap B$, and π_1, π_2 are the projections (see Figure A.10 (a)). It is easy to verify that the diagram A.10 (b) commutes.

Moreover, if there exists another pullback candidate, e.g., $A \xrightarrow{z_1} Z \xleftarrow{z_2} B$, where, e.g., $Z = \{3\}$ such that $f(3) = g(3)$, then there exists a universal (injective) mapping $h : Z \to A \cap B$ such that the diagram commutes. □

Example A.12. Let $A = \{1, 2, 3, 4\}$ and $B = \{3, 4, 5, 6\}$ be sets and let $C = \{1, 2, 3, 4, 5, 6\}$ be their intersection, i.e., there exist projections $f : C \to A$ and $g : C \to B$.

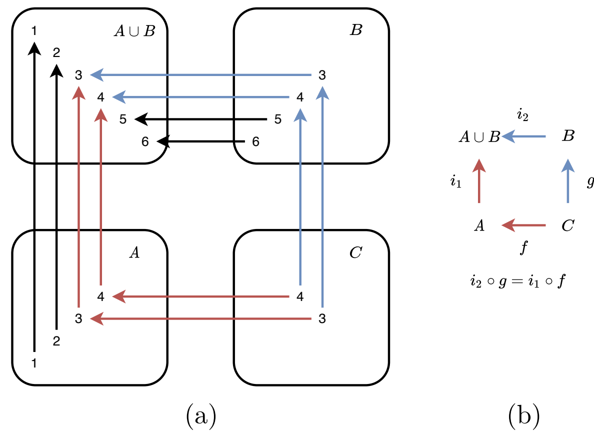
The pushout of $A \xrightarrow{f} C \xleftarrow{g} B$ is, e.g., $A \xleftarrow{i_1} Q \xrightarrow{i_2} B$, where Q consists of elements q such that $q \in A + B$, $A + B$ being a disjoint union of A and B and $a \in A$ are identical to $b \in B$ if there exists $c \in C$ such that $f(c) = a$ and $g(c) = b$, hence we obtain a union $A \cup B$ (see Figure A.11 (a)). It is easy to verify that the diagram in Figure A.11 (b) commutes. □

Finally, Table A.4 summarises the application of natural transformation in approaches representing data at the conceptual or logical level and in approaches performing data migration.



(a) (b)

Figure A.10: An example of a pullback corresponding to an intersection of sets



(a) (b)

Figure A.11: An example of a pushout corresponding to union of sets

Table A.4: Application of universal constructions

	Lippe and Ter Hofstede [18]	**CGOOD** [19]	**Spivak et al.** [20]	**APG** [21]	**MM-(evo)cat** [3, 5]
Product (Definition 13)	Complex identifier; Set	-	Querying (not discussed)	Querying (not discussed)	Complex identifier
Coproduct (Definition 14)	-	-	Querying (not discussed)	Querying (not discussed)	Multiple identifiers
Complemetable morphism (Definition 15)	Inheritance	-	-	-	-
Pullback (Definition 16)	-	-	Querying (not discussed)	Querying (not discussed)	Joining of data
Pushout (Definition 17)	Generalisation	Addition of data	Querying (not discussed)	Querying (not discussed)	Addition of data