FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Jáchym Bártík

Evolution Management in Multi-Model Databases

Department of Software Engineering

Supervisor of the master thesis:	Doc. RNDr. Irena Holubová, Ph.D.
Study programme:	Computer Science
Study branch:	Software and Data Engineering

Prague 2022

I thank Irena Holubová for the supervision of the thesis. I would especially like to highlight the patience and dedication with which she guided me, supported me and gave me invaluable feedback on the work, as well as the best advice.

Pavel Koupil helped me enormously with understanding the categorical approach and all the technical details of the MM-cat framework, for which I owe him my thanks. The discussions I had with him and Irena contributed greatly to this thesis.

I would also like to express my gratitude to my family for the unconditional support they always provided me, especially during the final stages of finishing this thesis. I promise I will fix my sleep schedule from now on.

Last but not least, I would like to thank my colleague, Štěpán Stenchlák, for countless cases of help and encouragement throughout my entire master's degree.

Title: Evolution Management in Multi-Model Databases

Author: Bc. Jáchym Bártík

Department: Department of Software Engineering

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

Abstract: Multi-model databases allow us to combine the advantages of various data models by storing different types of data in different models. However, this technology is still relatively immature, lacks standardization, and there are not any tools that would allow us to model multi-model data or manage their evolution. This thesis (i) provides an introduction to MM-cat, i.e., the framework for modeling multi-model data, (ii) describes the implementation of the framework, (iii) designs a workflow and a set of schema modification operations to facilitate evolution management and (iv) performs experiments to prove their reliability and scalability.

Keywords: Multi-Model Database, Evolution Management, Data Modeling, NoSQL, Category Theory

Contents

Introduction

With the emergence of Big Data and the challenges its processing presented, the NoSQL databases began to appear in order to find the best solution. The variety aspect of the data causes that any database model can not handle it effectively on its own. Instead, several models were developed and combining various database systems with possibly different models has become a common practice in present software development.

Today, all the most used database systems are multi-model, meaning that, to be able to take advantage of this technology fully and effectively, we have to be able to model the multi-model data first. The next natural step is evolution management, i.e., the ability to dynamically update the schema while the changes are automatically propagated to the data. Of other closely related challenges, we would mention querying over multi-model data and inference of the schema from the data.

In this thesis, we propose an approach based on category theory that aims to solve the issues of modeling and evolution management of multi-model data. It is both general enough to cover all existing data models and easily extensible to provide a basis for other features, e.g., querying and schema inference.

Thesis Outline

In Chapter 1, we provide a more detailed introduction to the target topic, as well as definitions of supported data models.

Chapter 2 contains an overview of existing approaches and a comparison of their features.

Category theory, forming the core of the proposal, is introduced in Chapter 3. We introduce the necessary mathematical structures and formalism.

In Chapter 4, we present the key concepts the proposed approach is based on. We describe its architecture, the workflow of data transformations, as well as important design decisions.

The creation and edition of a multi-model schema are performed using the so-called schema modification operations. We define them in Chapter 5.

As a proof of concept, we implemented a multi-model data modeling framework MM-cat. Its description forms the content of Chapter 6. MM-evocat, an extension to the framework enabling evolution management, is already being implemented.

Lastly, in Chapter 7, we present the results of performance experiments with the operations in order to prove their reliability and linear scalability.

1. Motivation

A typical application can not function without a place to store its data. The traditional way is to choose one of the most popular relational DBMSs. However, cases where this approach fails have recently started to appear. For instance, some applications need to store so large, varied, and constantly changing data that the relational databases can not keep up with.

In response to these requirements, new database models (such as the document, graph, etc.) were introduced, marking the beginning of the era of NoSQL databases. Although they solved the big data issues, they brought their own problems as well. The relational databases are standardized and well researched, which are the two most significant features that the NoSQL ones lack.

In an attempt to solve one of the so-far neglected drawbacks, this thesis focuses on the evolution management of the NoSQL databases.

1.1 Requirements on Modern DBMSs

The big data are denoted by several so-called V-characteristics [13], conveniently starting with the letter "v":

- *volume* – the data can be really huge, usually starting in petabytes,
- *velocity* – the data is generated in a much higher rate than the traditional data ever was,
- *variability* – the data come from many different sources, it combines multiple data formats etc.,
- *veracity* – the data is likely to contain errors and redundancy or to originate from doubtful sources.

These properties result in the following requirements for the modern DBMSs:

- *horizontal scalability* – we have to be able to store and process the data on multiple server nodes because no physical computer can be vertically scaled to keep up with the volume and velocity of the data,
- *redundancy* – the same piece of data needs to be served to multiple users at the same time, so we need to ensure we have enough copies of it,
- *fault tolerance* – the system has to be able to recover from failures such as a node outage or a division of the network (this is partially covered by the redundancy feature).

These features are not as easy-to-implement as they might seem. The main problem lies in the CAP theorem [5], which states that at most two of the following three properties can be achieved at the same time by one database system:

- *consistency* – a read request to any node returns the value from the node that was last written to,

- *availability* – each request to a non-failing node results in a response in a finite time,
- *partition tolerance* – any number of messages sent between the nodes can be delayed or lost.

The traditional relational databases focus on the first two conditions because they are run on a single node each, so they can freely ignore the last one. This allows them to guarantee the ACID properties. With the onset of decentralization in the NoSQL databases, we have to give up one of the first two conditions.

The most common approach is to keep availability (most of the time) and replace the (strong) consistency with a much weaker consistency model, usually the *eventual consistency*. The new set of guarantees is called BASE. The critical point is that in most cases, the behavior of the DBMS is not really that different. The system is available almost always, and the data should be consistent just a short time after they are written. Of course, this approach is not applicable in environments that require precision and do not tolerate errors. But the vast majority of use cases are not that critical.

1.2 Current Solutions

This section overviews the currently used database technologies and how they are applied to solve the challenges mentioned above.

1.2.1 Database Models

The most common database models can be divided into aggregate-oriented (key-value, column and document) and aggregate-ignorant (relational and graph). There are many others (namely RDF or array), but we will not cover them in this thesis.

Aggregate-Ignorant Models

First, we will mention the relational model since it is the most well-known, and all others can be explained by how they differ from it. The basic unit of data in the model is a relation. It is a subset of the Cartesian product of the domains (the sets of all possible values) of several attributes.

Let us consider an example. We have three attributes, A_1, A_2, A_3. The first one is an `id` with possible values from the field of natural numbers. So the domain of A_1 are the natural numbers, i.e., $D(A_1) = \mathbb{N}$. The second one is a `name` whose values are strings of English letters. The last one is a `datetime` whose values are the Unix timestamps. Any tuple with exactly these three values is an element from the set $D(A_1) \times D(A_1) \times D(A_3)$. The relation is a set of all the elements of this kind we have in our database. It can be represented as a table with the attributes as the columns and the tuples as the rows.

A set of attributes that uniquely identify each tuple of a given relation is called a superkey. Because the tuples must be unique, a tuple is trivially its own superkey. On the other hand, a key (primary key, candidate key, identifier) is a superkey from which no attribute can be removed while still remaining the superkey. A database usually has multiple relations. The neat part is that there is a sophisticated mathematical theory that allows us to join the relations together to produce new views on the data. For instance, an attribute from one relation can represent a key from another and can be used to connect the two rows.

Good practices for relational databases include minimizing redundant data by referencing the data with keys instead of storing it by values. This process is called normalization. The relational databases support many types of integrity constraints and usually enforce strict types. They also require a schema to be specified before the data is inserted – so-called schema-full or schema-on-write approach.

One of the main advantages of this model is its atomicity. Multiple simple operations can be grouped into one transaction. They can even span across multiple rows from various tables. A transaction is guaranteed to execute all or none of its operations.

The second type of aggregate-ignorant model we cover is a graph database. It consists of nodes and edges. A node has an identifier, a set of properties (usually key-value pairs) and a set of labels that allows us to group similar nodes. For example, some nodes can have the label `user` while a subset of them will also have the label `admin`. An edge is a directed relationship between two nodes. It has its own identifier, the identifiers of the nodes, a set of properties and a label.

Although the graph model can be simulated by a relational database, its main advantage is that a dedicated graph engine can run specific queries much more effectively. The relational databases are best suited for querying many elements at once while joining only a few tables. On the other hand, graph databases excel in querying data for one entity through a large number of nodes and edges. They are usually used for searching patterns – e.g., when we need to find all the friends of the friends of a selected user that share a specific set of properties.

Aggregate-Oriented Models

Unlike the previous group, these models have a basic unit of data called *aggregate*. The atomicity of each transaction is guaranteed only within the chosen aggregate. They are also not suitable for storing relationships. A common approach is accepting redundancy and putting all the relevant data into the aggregate together. For instance, in the document model, it usually does not make much sense to have one document for a `customer` and another for the `address`. Even though the `address` might be reused by all of the `customers`'s `orders`, the preferred approach is to copy it everywhere it is needed.

The key-value model is represented as a map of the unique identifiers to the records, i.e., the aggregates. Although the record can have an internal structure in some key-value databases, it still has to be only a simple one, for instance, an atomic value, a tuple, or a collection. There is usually a support for additional features such as a timer, individual for each record, that deletes it after a predetermined time.

The column model uses a similar structure as the key-value one. It is a map of the unique identifiers to the records. The difference here is that the records have a more complex structure – each one is a map of the column names to the values (which can, again, be atomic values, in some systems also tuples or collections). The column names can be specified in advance (or have to, in some database systems), which is an example of a schema-mixed approach. So the columnar database resembles a table from a relational database where all of the columns are optional and can be created dynamically.

Lastly, the document model uses the same principle again – the key identifies the record. In this case, the record is a hierarchical document. It usually has a well-known type, i.e., JSON or XML. Unlike in the previous two models, the hierarchy can contain any level of nesting. Thanks to this feature, we can query inside the hierarchy.

As mentioned above, the documents prefer nesting instead of referencing other documents. Nevertheless, the referencing is definitely possible and, in some cases, even necessary (i.e., for recursive structures).

1.2.2 Multi-Model Data

The discussed models have their advantages and disadvantages, reflecting the expected use cases. However, a typical application uses various types of data, each of which would benefit the most from a different database model. This is, of course, possible, but such idea is not easy-to-implement. Currently, there exist two contradictory approaches.

The first option is to use the *polyglot persistence*. This means that the application uses multiple database systems at once. Since the direct usage of different databases is likely to lead to serious dependency issues, we usually create a *mediator* first. This software lies on top of the databases and provides one unified API for the application.

The disadvantages of this approach are numerous. We have to deal with many different database systems, i.e., we have to know their specific query languages, configurations, etc. In addition, the integration of these systems brings a whole new set of problems – writing queries for multiple databases at once, synchronization of their data, transactions, optimizations and so on.

The *multi-model* databases partially solve these issues by allowing multiple different models (e.g., relational, document, key-value, graph, . . .) to coexist within one database engine. There are multiple ways how to combine these models, for example:

- one model can be embedded in another,
- the entities from one model can reference the entities from another,
- redundancy, i.e., the same data can be stored in multiple models.

Currently, most of the popular database systems are multi-model. However, this means only that they support at least two database models. Actually, none of them supports all of the five models at once [12].

1.3 Challenges of Multi-Model Data Management

In the previous section, we discussed the state-of-the-art solutions for the modern database requirements. Now we will mention some of their most significant drawbacks. The challenges they represent are all the more difficult because the multi-model databases combine multiple models with often contradictory features, for example:

- schema-less vs. schema-full vs. schema-mixed,
- BASE vs. ACID,
- flat data with references vs. nested objects with redundancy, ...

1.3.1 Standardization

Various multi-model databases implement different sets of database models. Moreover, they often do so in divergent ways. There is neither a common query language nor a strategy for executing statements. In other words, the multi-model databases lack standardization.

We can take an example from the relational databases. Although each of the major relational databases has some specifics, they all are compatible with the SQL standards [7]. So, it is generally easy to migrate the data from one database system to another. On top of that, we can apply the knowledge and skill we learned for one system to another. Unfortunately, most of this is not possible in the multi-model world [12].

1.3.2 Modeling and Evolution

Now we come to the key issue of this thesis. Although most of the above-discussed data models are schema-less, the application that uses them has to work with a concrete schema. Hence, there is a need for a tool capable of modeling a multi-model schema. The more universal it would be, the better.

This tool would serve as a foundation for many other features. One of them is evolution management. This means that whenever we edit the schema, the system would automatically update the data to be compatible with the new schema. In the relational world, there exist migration software that can detect changes even in the application code and update the database accordingly. Another promising feature is schema inference. The tool should be able to find a structure in any given schema-less data and infer a schema based on it.

Lastly, the modeling tool would allow us to solve the issues mentioned in the previous section. Queries across multiple models will be naturally supported if we have one universal schema that encapsulates all the models. Further, data migration between multiple database systems is just a necessary prerequisite to evolution management.

2. Related Work

In this chapter, we describe popular evolution management tools that inspired our proposed approach. Each of them provides a unique view on the issues of evolution management. In the last section, there is a comparison of them and a summary.

2.1 Categorical Approaches

The two following approaches are based on the category theory. However, they use it at a level of complexity far beyond the scope of this thesis, so we do not cover them in detail as we would need to introduce a complex theoretical background first. Instead, we focus on the supported SMOs and overall architecture.

The first one is introduced in paper [19] by Tuijn and Gyssens from 1996. They propose a data model, represented by a typed graph, called CGOOD (*Categorical Graph-Oriented Object Database Model*). Both the model and the data are defined only by constructs from category theory. All SMOs can then be expressed in the form of functors.

The authors define *single addition*, *full addition*, *single deletion* and *full deletion* operations. All of them use pattern matching to find the respective part of the graph (in the case of the *single* ones) or all corresponding parts (in the case of the *full* ones) and then update them accordingly.

The second approach, proposed by Spivak et al., introduces FQL (*Functorial Query Language*) in paper [16] from 2015 and later CQL (*Categorical Query Language*) [1]. The approach is still being actively developed, the latest paper [14] was published in 2022.

In [16], the authors also use functors to define the SMOs: *rename*, *delete*, *copy*, *intersection* and *union*. All of them are available both on the *kind*[1] and the *property*[2] level.

2.2 Darwin

Darwin is a platform for schema evolution management in various NoSQL databases. It is also capable of migrating data between different databases. However, it does not support schema evolution of multi-model data.

The platform focuses on versioning. It keeps track of all SMOs, so, thanks to their reversibility, it is able to jump to an arbitrary version of the schema. The schema (including its history) can be extracted from the data itself. This way, the platform can detect changes in the incoming data and automatically produce the respective SMOs. We also have the option to trigger them manually.

[1]Kind is the unifying term for the independent, top-level data entities such as tables (in a relational database), collections (in a document one) and so on.

[2]Similarly, term property unifies the data entities that depend on a kind. They are usually called attributes (relational), fields (document), etc.

The schema-less databases naturally support the coexistence of data with different schemas. Therefore, Darwin is able to keep different parts of the data in different versions. However, it can still be queried as a whole thanks to *query rewriting*. This allows a variance of migration strategies:

- *Eager* – the SMOs are executed immediately. An advantage of this strategy is that all data is always actual, so we can fetch it with the lowest latency possible. We also do not have to worry about inconsistencies between different versions. On the other hand, performing all SMOs at once might cause a non-trivial downtime.
- *Lazy* – the SMOs are not applied until it is needed. For example, whenever we fetch data of a particular entity, it is automatically updated by executing all pending SMOs. This approach minimizes the total time we spend performing SMOs. However, it can lead to a cascading migration, i.e., a situation when multiple objects depend on each other, so we have to update all of them at once, causing a spike of latency in the process.
- *Hybrid* – a strategy that combines both of the previous approaches. The authors proposed two of them:
  - *Incremental* – most of the time it is the *lazy* migration. However, once in a while (for example, after a certain number of SMOs), the whole data is updated at once.
  - *Predictive* – the most frequently accessed data uses the *eager* strategy while the rest is updated according to the *lazy* one.

In [15], the authors introduce, among other ideas, their version of a declarative NoSQL schema evolution language that defines the SMO operations used by the Darwin platform. The first three, *add*, *delete* and *rename*, works on a single property of a kind or on the kind itself. The other two, *copy* and *move*, transfer on or more properties between two kinds.

2.3 MM-evolver

The authors introduce MM-evolver [6], a prototype of a multi-model data evolution tool. It is inspired by Darwin in the sense that it uses the same operations (i.e., *add*, *delete*, *rename*, *copy* and *move*). The authors also introduce their own NoSQL schema evolution language, called *MM SEL* (*Multi-Model Schema Evolution Language*).

However, unlike Darwin, the MM-evolver supports multi-model data. The operations can transfer kinds, properties and data between different models. The *MM SEL* language is executed in the model-independent layer. For each operation, the engine finds out which models it affects and translates it to their model-specific languages. All data is then immediately updated, which corresponds to the *eager* strategy.

The tool also focuses on reference evolution – whenever a property is deleted, moved, etc., all references to it are automatically updated accordingly. These changes also propagated between models. A new operation, called *reference*, is introduced for creating the references.

2.4 Hernández Chillón et al.

In [3], the authors propose a taxonomy of *schema change operations* (i.e., SMOs), which are defined on the so-called *U-Schema* [2]. They also introduce a domain-specific language, called *Orion*, that implements them. The taxonomy contains not only the basic operations mentioned in the previous two sections but also many others.

For example, the kinds can also be split or merged. Properties can be nested, unnested, cast (to another type), promoted (added to a key), and so on. There is also support for the so-called *variations*, i.e., different versions of the same kind.[3] These database-independent operations are then translated to the database-specific ones and executed.

Although the *U-Schema* supports multiple types of NoSQL databases (i.e., key/value, columnar, document and graph) and partially the relational ones, it does not unify them in the sense of multi-model data. It integrates various model-specific concepts into one schema instead of building an abstract layer above them that would allow a genuinely unified approach.

2.5 Comparison

Now we compare all the previously described approaches, as well as the tool MM-evocat, which will be introduced in this thesis. An overview is displayed in Tables 2.1 and 2.2, which were taken from [8].

The first one compares various features of the approaches described above. As we can see, each of the approaches introduces both its own schema and a set of operations to modify it (and, of course, the underlying databases). All of them, except Darwin, support only eager migration. The same holds for version jumps and query rewriting, which both follow from the ability to update data lazily. Both Darwin and Orion can infer the schema from the data. Again, thanks to its focus on versioning, Darwin can also infer the history of the schema.

Generally, the more recent the approach is, the more different models it supports. However, the ability to represent multi-model data, which is crucial for this thesis, is provided only by MM-evolver. The SMOs, depicted in the second table, follow the same path as the models – the later the tool is, the more of them it defines.

[3]Unlike in the Darwin platform, the versions do not capture changes over time. Instead, they all exist simultaneously. We can view them as different subclasses that can all be assigned to the variable with the type of their common ancestor.

Table 2.1: A comparison of features in the selected evolution management approaches.

	Tuijn and Gyssens [19]	**Spivak et al.** [16]	**Darwin** [17]	**MM-evolver** [6]	**Hernández Chillón et al.** [3]	**MM-evocat** [10]
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 2.2: A comparison and classification of supported SMOs in the selected evolution management approaches.

	Tuijn and Gyssens [19]	**Spivak et al.** [16]	**Darwin** [17]	**MM-evolver** [6]	**Hernández Chillón et al.** [3]	**MM-evocat** [10]
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	add_id, drop_id
Reference-level	-	-	-	reference	morph	add_ref, drop_ref
Cardinality-level	-	-	-	-	mult	changeCardinality
Version-level	-	-	-	-	delvar, adapt, union	-

3. Theoretical Background

The proposed approach is based on *category theory* (see Section 3.1) which is abstract enough to contain all currently used data models. The necessary theoretical background is described in paper [9]. A brief summary of the most important concepts used explicitly in this thesis is provided in this chapter. Note that there are categorical constructs that we use implicitly. However, for use-friendliness, we keep the formal background easy to understand.

The generality of the theory allows us to create a category that describes multiple logical models from different databases at once. Thus, we are able to model any combination of different logical schemas with one category. This object is called *schema category* (see Section 3.2).

The next step is to import data from all the databases to a unified representation called *instance category* (see Section 3.3) and to export the data from the instance category to specific databases.

The advantage of this approach is obvious – we can convert the data from database **A** to any other database **B** (with possibly different data model) simply by transforming them from **A** to the instance category and from there to **B**. We only need to create a wrapper for each involved database system that would handle the platform-specific requirements of the given database. Otherwise, we would have to create a specific converter for each pair of supported database systems.

Each logical model is mapped to the schema category via its own *mapping* (see Section 3.4). This structure defines both the mapping and the logical model itself.

3.1 A Brief Introduction to Category Theory

A *category* consists of a set of *objects*, a set of *morphisms* and a binary *composition operation* over the morphisms. We can view it as a directed multigraph – the objects are the nodes and the morphisms are the directed edges.

The operation allows us to combine morphisms. If we have morphism f, i.e., an edge from object A to object B (which can be written as $f\colon A \to B$), and another morphism $g\colon B \to C$, we can combine them to the morphism

$$g \circ f\colon A \to C\,. \tag{3.1}$$

The whole category is closed under this operation. Also, it must hold that chaining of these operations is associative, for example,

$$(h \circ g) \circ f = h \circ (g \circ f)\,. \tag{3.2}$$

Another requirement is that there is an *identity morphism* $i_A\colon A \to A$ for each object A. The identity law must be satisfied, so

$$i_B \circ f = f\,, \tag{3.3}$$

$$f \circ i_A = f\,. \tag{3.4}$$

$$\begin{tikzcd}
A \arrow[loop above, "i_A"] \arrow[r, "f"] \arrow[rr, bend right, "g \circ f"'] & B \arrow[loop above, "i_B"] \arrow[r, "g"] & C \arrow[loop above, "i_C"]
\end{tikzcd}$$

Figure 3.1: An example of a category.

An example of a category is displayed in Figure 3.1.

The direct morphisms (i.e., from one object to a different one) are called *base morphisms*. Those created by composition are called *composite morphisms*. The category contains all possible composite morphisms. However, this is really inefficient for practical purposes. During the implementation, we decided to create them lazily only when needed. So whenever we say "*create a composite morphism*" in this thesis, we mean it purely in the sense of the specific technical solution instead of category theory.

3.2 Schema Category

Any application needs a way to represent the data. A conceptual schema is a high-level model of the data from the application's point of view. It hides the technical details of physically storing the data from the application and allows us to concentrate on the concepts essential for the domain we are working with. In the proposed approach, this model is represented by a schema category. A similar structure capable of holding concrete data is called the instance category.

On the other hand, a logical model describes how the data should be organized into tables (for the relational model), documents (for the document model) and so on. It generally does not depend on any specific database engine. However, it might be affected by the possibilities of the target type of database systems.

This definition of the conceptual and logical model corresponds to the PIM and PSM from MDA. We do not consider the CIM because it is beyond the scope of this thesis. Also, we use the terms *model* and *schema* interchangeably.

Our approach is based on the condition that an application should have one conceptual schema which can be mapped to multiple logical models. The models can partially or entirely overlap, and each of them can be suited for a different database system.

A schema category is a category, which means that it is a tuple of objects, morphisms and a composition operation over the morphisms.

3.2.1 Objects

Each schema object corresponds to a kind or a property from the conceptual schema. It is a tuple of:

- *key* – an internal identifier,
- *label* – a human-readable identifier,
- *superid* – a set of data attributes the object is expected to have,
- *ids* – a set of all identifiers of the object (each one of them is called *id*, plural *id*s).

Each attribute of *superid* is represented by a morphism that goes from the object with *superid* to the object corresponding to the attribute. As will be shown later, a morphism is identified by its *signature*. So *superid* is just a set of *signatures*.

An identifier is also a set of *signatures*, this time their morphisms lead to the objects that together unambiguously identify the given object. Hence, *ids* is a set of sets of *signatures*. It must hold that all the *signatures* from all the sets in *ids* are also contained in *superid*.

The most simple schema object is an object with only one identifier (the *empty signature* ϵ) and a single element *superid* (also the *empty signature*). So its *ids* and *superid* are $\{\{\epsilon\}\}$ and $\{\epsilon\}$ respectively. This means the object represents a simple property identified by its value. More complex objects are identified by and contain values of other objects.

3.2.2 Morphisms

A morphism represents a directed edge from one object to another. It is modeled as a tuple of:

- *signature* – an internal identifier,
- *domain object* – an object in which the edge starts,
- *codomain object* – an object in which the edge ends,
- *cardinality* – one of the following: `0..1`, `1..1`, `0..*`, `1..*`.

For simplicity, the *signatures* of base morphisms are represented by integers. The convention is that if a morphism has a *signature* n, its dual has a *signature* $-n$. The identity morphisms are represented by the empty *signature* ϵ.

The *signatures* can be concatenated as strings with a dot as a separator. For example, consider objects A, B and C with morphisms $f\colon A \to B$, $g\colon B \to C$ and corresponding *signatures* $\sigma_f = \texttt{5}$, $\sigma_g = \texttt{-7}$. If we combine them to the morphism $g \circ f\colon A \to C$, its *signature* will be

$$\sigma_{g \circ f} = \sigma_g \circ \sigma_f = \texttt{-7.5}\,. \tag{3.5}$$

The domain and codomain objects are represented by their identifiers, i.e., *keys*.

The current version of the framework requires that for each morphism that is not an identity morphism, there is an opposite morphism (called *dual*). For example, if there is a morphism $f\colon A \to B$, there must exist a dual morphism $\overline{f}\colon B \to A$. Because of it, we can consider the combination of both of them as one two-way relation. For instance, imagine $f\colon A \to B$ and $\overline{f}\colon B \to A$. If their cardinalities are c a $\overline{c}$, we denote the cardinality of the combined relation as $(\bar{c} \leftrightarrow c)$. Because each morphism has one of the four cardinalities, there are 16 distinct combinations for the whole relation.

3.2.3 Example

This example is taken from paper [9]. An ER schema is depicted in Figure 3.2. The color borders define which entities are stored in which type of database:

- green – document,
- red – columnar,
- yellow – key-value,
- purple – relational,
- blue – graph.

Examples of the respective data are in Figure 3.3. A schema category to which the ER schema is transformed is shown in Figure 3.4. This transformation can be done automatically [18]. The schema category can also be created directly without the need for an ER schema by a tool described in Chapter 6.

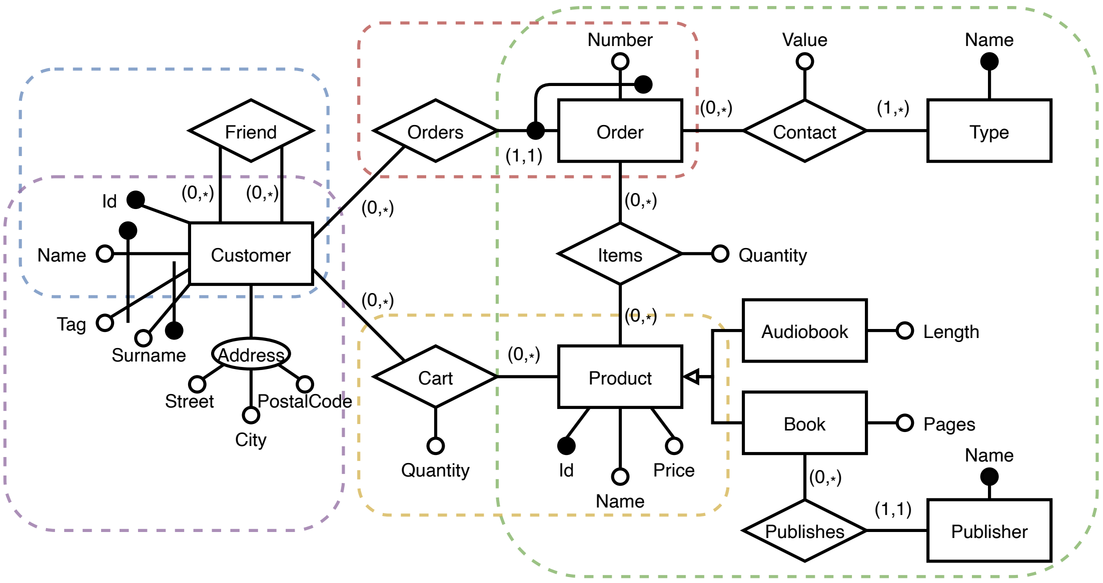
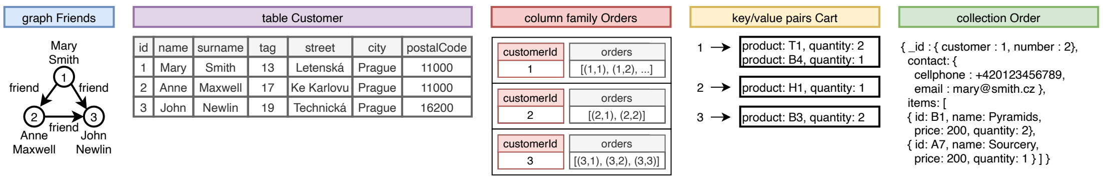


Figure 3.2: An example of an ER schema.



id	name	surname	tag	street	city	postalCode
1	Mary	Smith	13	Letenská	Prague	11000
2	Anne	Maxwell	17	Ke Karlovu	Prague	11000
3	John	Newlin	19	Technická	Prague	16200

customerId	orders
1	[(1,1), (1,2), ...]
2	[(2,1), (2,2)]
3	[(3,1), (3,2), (3,3)]

```
1 → product: T1, quantity: 2
    product: B4, quantity: 1
2 → product: H1, quantity: 1
3 → product: B3, quantity: 2
```

```
{ _id : { customer : 1, number : 2},
  contact: {
    cellphone : +420123456789,
    email : mary@smith.cz },
  items: [
  { id: B1, name: Pyramids,
    price: 200, quantity: 2},
  { id: A7, name: Sourcery,
    price: 200, quantity: 1 } ] }
```

Figure 3.3: Different database models for various logical models that can be mapped to the ER schema, which is depicted in Figure 3.2.

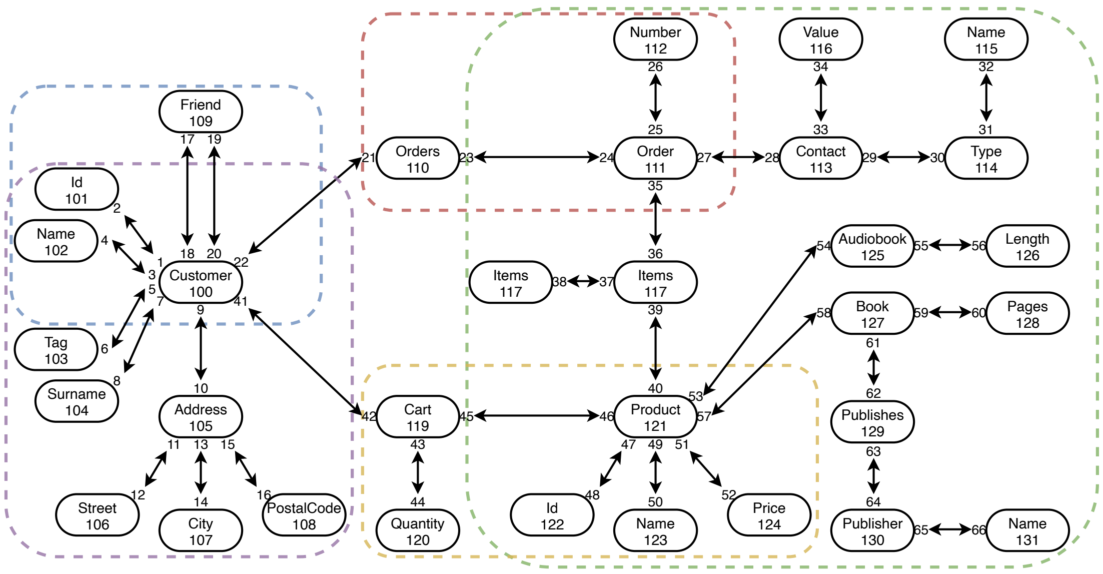


Figure 3.4: An example of schema category created from the ER schema, which is depicted in Figure 3.2.

3.3 Instance Category

An instance category is a structure capable of holding actual data. It is organized in the same way as the schema category, however, its entities have different contents.

3.3.1 Objects

Consider a schema object A with *superid* $\{\sigma_f = 3, \sigma_g = 6\}$ where $f\colon A \to B$ and $g\colon A \to C$ reference objects B and C. Let us denote t_i as one specific instance of values of these objects. Then the corresponding instance object can be considered as a set of all t_i in our domain. One t_i can be represented as a set of tuples (σ, v), where σ is one of the *signatures* and v is a value from the domain of the object linked to A with morphism with *signature* σ. So in this case, t_i can be written as

$$t_i = \{(\sigma_f, b_j), (\sigma_g, c_k)\}\,, \tag{3.6}$$

where b_j is one of the possible values of B and c_k is the same for C.

For example, let us consider that we are modeling Middle Earth and object A represents the characters. They have to have a first name (object B with possible values `Frodo`, `Bilbo`, `Samwise`) and a last name (object C with possible values `Baggins` and `Gamgee`). With these values available we are able to create the characters (i.e., instances of A)

$$t_1 = \{(3, \texttt{Frodo}), (6, \texttt{Baggins})\}\,, \tag{3.7}$$
$$t_2 = \{(3, \texttt{Bilbo}), (6, \texttt{Baggins})\}\,, \tag{3.8}$$
$$t_3 = \{(3, \texttt{Samwise}), (6, \texttt{Gamgee})\}\,. \tag{3.9}$$

Because B and C are simple properties, they are identified by the empty *signature*.

So the possible instances of C would be

$$t_1 = \{(\epsilon, \texttt{Baggins})\}, \tag{3.10}$$
$$t_2 = \{(\epsilon, \texttt{Gamgee})\}. \tag{3.11}$$

One t_i is called an *active domain row.* A set of all t_i together creates an *active domain* (also called *instance object*) of the schema object A, which is denoted as $I(A)$.

It is important to note that the objects referenced in t_i must be the simple schema objects with the empty *signature* as an identifier, such as B or C. A complex object (e.g., A) can also be referenced, but we have to use one of its identifiers to do so.

3.3.2 Morphisms

An instance morphism is a set of all relations between the rows from two instance objects. If $f\colon A \to B$ is a schema morphism with the domain object A and the codomain object B, we can denote m_i to be a specific relation between selected two rows $a_j \in I(A)$ and $b_k \in I(B)$. In other words, m_i is an ordered pair $m_i = (a_j, b_k)$ and is called *active mapping row.* The instance morphism of f, denoted as $I(f)$, is then the set of all possible m_i.

If we go back to the previous example, all $m_i \in I(f)$ would look like

$$m_1 = (\{(\texttt{3}, \texttt{Frodo}), (\texttt{6}, \texttt{Baggins})\}, \{(\epsilon, \texttt{Frodo})\}), \tag{3.12}$$
$$m_2 = (\{(\texttt{3}, \texttt{Bilbo}), (\texttt{6}, \texttt{Baggins})\}, \{(\epsilon, \texttt{Bilbo})\}), \tag{3.13}$$
$$m_3 = (\{(\texttt{3}, \texttt{Samwise}), (\texttt{6}, \texttt{Gamgee})\}, \{(\epsilon, \texttt{Samwise})\}). \tag{3.14}$$

3.4 Mapping

Each kind from the conceptual schema needs to be mapped to a logical model representing data in the selected database system. A mapping uses a JSON-like structure called *access path* to both define the logical model of the kind and specify how it is mapped to the conceptual model.

Mapping also defines in which database the data of the kind is stored. It is a tuple of:

- *database* – type and connection credentials,
- *name* – the name of the kind in the database,
- *root* – the object from the schema category associated with the root node of the access path,
- *pkey* – a (possibly ordered[1]) collection of *signatures* of morphisms whose codomain objects form the *primary identifier* of the kind,
- *ref* – a set of references from the kind,
- *access path.*

[1]If the logical model requires so.

In the original version [9], there was also a *root morphism* because it was possible to associate the kind with a morphism in addition to an object. This property could have been undefined.

3.4.1 Access Path

An access path is a tree. The root node represents the kind of the mapping. Each other node corresponds to one or no object from the schema category. The nodes that do not correspond to any object are called auxiliary.

Each edge of the access path corresponds to a morphism between the objects from the schema category to which the two nodes connected by the edge belong. We can represent it by putting the *signature* of the morphism to the child node. However, an edge between a regular node and an auxiliary one does not correspond to any morphism. Therefore, the auxiliary node gets assigned a *null signature*.

For example, let us consider objects A and B with morphism $f\colon A \to B$ and respective nodes n_A and n_B. Now we create an access path starting with node n_A which has a child node n (an auxiliary node) with a child node n_B. Node n will have a null *signature*, n_B will have the *signature* of the morphism f, i.e., σ_f. Now we can find the *signature* of morphism from A to B by simply concatenating *signatures* from all nodes on the path from n_A to n_B (including n_B). The *null signature* acts effectively as an *empty* one, so we get $\epsilon \circ \sigma_f = \sigma_f$.

3.4.2 Names

The last thing a node needs to be able to define a mapping is a name. For a relational database, this means the name of the column. For a document one, it is the name of the property etc. We consider three types of names:

- *static* – a string,
- *anonymous* – an empty name. It has two use cases:
  - the root node of the access path,
  - a node that represents complex objects in an array,
- *dynamic* – the name of the property corresponds to the value of another object from the schema category.

The last one might be a little confusing, but it is just a way how to represent a map. For example, consider property `contact` with multiple subproperties like:

```
contact: {
    email: "example@example.com",
    phone: "123456789",
        ...
}
```

The data structure is dynamic, meaning that one `contact` has only `email`, another has both `email` and `phone` while a third one has something entirely different. But we have no way of knowing what are all the possible names these subproperties have. So we have to map both their values and their names at the same time.

3.4.3 Representation

The access path can be represented by a JSON-like structure. For example, let us consider a kind `user`:

```
{
    name: 1,        // Simple property
    address: 3.2 {  // Complex property
        ...
    },
    contact: 4 {
        6.5: 7      // Property with a dynamic name
    }
}
```

The first line defines a simple property with static name `name` and *signature* `1`. The second line describes a complex property `address` with *signature* `2.3`. This property can contain any number of other properties, for example `street`, `city` and `ZIP`. Lastly, `contact` is a complex property that contains subproperties with dynamic names. Note that those subproperties have to be contained in `contact` property because they would not be recognizable from other properties (e.g., `name`) otherwise. The morphism with *signature* `6.5` defines the name of the property while `7` determines its value.

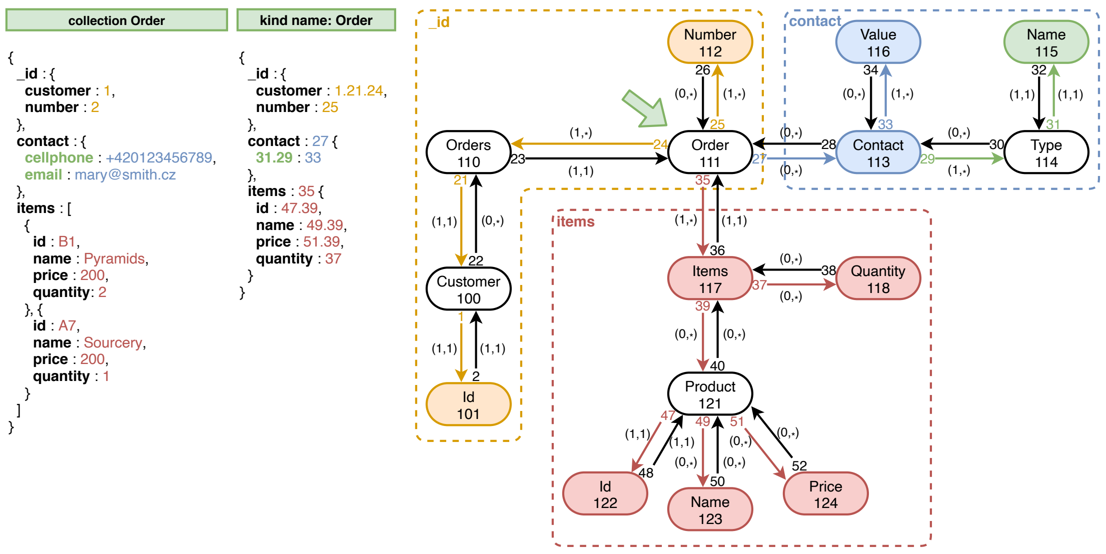
Also, be aware that the cardinalities of the properties are not defined explicitly because they can be derived from the cardinalities of the morphisms instead. So if morphism `3.2` has cardinality `0..*` or `1..*`, the property is an array.

3.4.4 Example

We conclude with the example from the previous section. Let us focus on the schema in Figure 3.5, which is a part of the larger schema category depicted in Figure 3.4.

The JSON-like structure in the middle of Figure 3.5 represents an access path of kind `Order`. Unlike in Figure 3.2, the colors now represent different groups of schema objects that are mapped to the top-level properties of `Order`.

Lastly, we can see data of these objects on the left, corresponding to the first entity of kind `Order` from Figure 3.3



collection Order

```
{
  _id : {
    customer : 1,
    number : 2
  },
  contact : {
    cellphone : +420123456789,
    email : mary@smith.cz
  },
  items : [
    {
      id : B1,
      name : Pyramids,
      price : 200,
      quantity: 2
    }, {
      id : A7,
      name : Sourcery,
      price : 200,
      quantity : 1
    }
  ]
}
```

kind name: Order

```
{
  _id : {
    customer : 1.21.24,
    number : 25
  },
  contact : 27 {
    31.29 : 33
  },
  items : 35 {
    id : 47.39,
    name : 49.39,
    price : 51.39,
    quantity : 37
  }
}
```

Figure 3.5: An example of an access path (middle) which maps kind `order` (right) to a document database (left).

4. The Proposed Approach

As we have shown in Chapter 1, although the multi-model data approach has great benefits, it also presents many challenges. We aim to contribute to their solution by utilizing the categorical representations introduced in Chapter 3, especially regarding data modeling, migration and evolution management. In this section, we introduce the proposed solution, i.e., a multi-model modeling and evolution management framework.

4.1 Architecture

As described in Section 3.2, we consider two levels of models, the conceptual and the logical. It is natural to divide the architecture of our approach into two layers. The first one contains the conceptual model, which is defined by the schema category (see Section 3.2) and represents the high-level, platform-independent structure of the data. On the other hand, the second layer contains multiple overlapping logical models. Each of them is customized for a specific database system – it is bound by the system's limitations. The logical models are defined by the mappings (see Section 3.4), which also map them to the conceptual model.

The changes appear in both layers. The user might need to change one of the logical models but keep the conceptual schema intact. An example of this use case would be when the user wants to transfer the data from one database system to another of the same type. In order not to edit the conceptual schema, the mapping between it and the logical ones must be updated.

Another use case is to change only the conceptual schema. Consider, e.g., a very complicated and mostly obsolete schema mapped to much simpler logical models. The schema was designed to be robust, but later the user finds out that this complexity is not needed. So he or she may want to simplify the schema but keep it corresponding to the models. The mappings must be updated accordingly, as in the previous example.

The last option is to change both the conceptual and the logical models. An example would be a simple addition of a new object to the schema followed by the creation and addition of a corresponding logical model.

4.2 Design Decisions

Here we present an explanation for several decisions we have made during the design and development of the framework.

4.2.1 Hierarchy of Models

The framework is designed to shield the user from the platform-specific details of the logical models. The user is expected to focus mostly on the conceptual schema, so the interactions with the logical models should be limited as much as possible. Therefore, most of the SMOs the user has an access to should originate in the conceptual model.

As shown above, some of these operations have direct consequences on the logical models, or they require the user to make additional changes to them. In all cases, the underlying principle is that the logical models are always dependent on the conceptual model. In our approach, the changes made in the conceptual model do propagate to the logical models, but the updates of the logical models do not affect it.[1]

Therefore, we have to design the SMOs to maximize the automation of their propagation to the other models. In an ideal world, the user would do only the operations on the conceptual layer while the system would handle the rest. On the other hand, we must make them powerful enough to allow the user to achieve any desired change. However, we should still keep them as simple and user-friendly as possible.

4.2.2 Lazy vs. Eager Migration

Lazy and eager migration, as well as two hybrid strategies, are discussed in Section 2.2. To sum it up, the main issue of eager migration lies in the fact that a mass update of the whole database might require so much time that the users might experience downtime. Another disadvantage is the total work needed to update the schema repeatedly.

On the other hand, the lazy approach (and all the hybrid ones) significantly increase the system's complexity. We would have to enable multiple versions of the same kind to exist simultaneously. We would also have to version everything, including the individual data entities. Last but not least, such a system would be much more prone to errors. Considering that the proposed framework should be a proof concept of the categorical Approach and not an optimized application, we decided to use the eager strategy. However, the lazy approach will be considered in future work.

4.3 Workflow

In this section, we describe the general workflow of evolution management. Each of the following subsections defines one step in this process. First, let us assume there is a schema category S, a set of all mappings M, and data D.[2]

The first two steps are (i) editing objects from S and (ii) fixing their corresponding mappings from M. They are supposed to be iterated as many times as needed – a change of an object might require a change of mapping and for the user, it is more convenient to gradually edit both. During the first two steps, D is not being updated. Then (iii) the system collects all SMOs, optimizes them and applies them to the instance category, i.e., the actual data D.

[1] More precisely, a change in a logical model might result in the need to edit the conceptual model. However, it is the user, not the system, who has to initiate the change. Although the other way (i.e., the logical to conceptual model propagation) is definitely feasible, we leave this idea to future work.

[2] Both S and M can be created manually using tools described in Chapter 6. Also, the schema category can be inferred from the data automatically by the MM-infer tool [11].

4.3.1 Step (i) – Editing of the Schema Category

Firstly, the user needs to specify the SMOs. An editor (an extension of the tool described in Chapter 6) will be used for this purpose. The result will be a list of operations ordered by when they are to be applied and a new version of the schema category, denoted as S'.

During this process, it may happen that the schema category gets to an invalid state.[3] That is perfectly fine, provided the schema category will be valid after the last operation. The user will be allowed to save the changes only if this condition is met. There are also operations that are required to originate from a valid state, so the user will not be able to use them otherwise.

4.3.2 Step (ii) – Propagation to the Mappings

Each SMO can affect the mappings in one of the following ways:

- it targets a mapping directly, i.e., it is performed at the logical level,
- it can be automatically propagated to the mappings, for instance, deleting a property from a kind,
- it can find the mappings it affects, but additional user input is required to update them, e.g., adding a property to an already mapped kind,
- there are no corresponding mappings, for example, when creating a new kind.

The user has to manually update and create all the mappings for the last two options. The result of this is a new version of the mappings M'.

4.3.3 Step (iii) – Propagation to the Instance Category

When the user is done with the first two steps, the system checks if the data can be updated directly from the changes in mappings ΔM. For example, renaming a column in a table from a relational database does not require any knowledge about the data – we can simply generate the corresponding `ALTER TABLE ...` statements. If that is the case, the system creates the data update statements ΔD and applies them to the data.

In most cases, the actual data needs to be updated, i.e., we need it to generate ΔD. So the system uses the current versions of S, M and D to create an instance category I. This part of the process is described in [9] as the *Model-To-Category Transformation* and we already implemented it.

Then, the SMOs are applied to the instance category (in the same way as they were applied on S in the first step – we use the fact that S and I have the same structure) to transform it into the new version I'. From it, the update statements ΔD can be exported using the new mappings M'. Finally, ΔD are applied to the data. The last part is, again, described in [9] as *Category-To-Model Transformation* and already implemented.

[3]For example, the user might want to add a kind identified by one of its properties. To specify the identifier, he or she would have to first create a morphism between both objects. However, to create a morphism, both its domain and codomain object must already exist.

4.3.4 Summary

A schema of the whole process is depicted in Figure 4.1. An advantage of this approach is that while the user can edit the schema and the mappings iteratively, we still have all the operations available before the data is updated. So we can analyze them to find possible optimizations and only after that run the potentially time-consuming transformation on the actual data. For instance, if the user creates a copy of an object and then deletes the copy later without ever interacting with it, we can cancel both operations. This concept is called *version jumps* in the Darwin framework.

$$\begin{tikzcd}
 & S,\, M \arrow[d, dash] \arrow[rrr, dashed, "\text{SMOs}"] \arrow[rrrrd, dashed, bend left=40, "\Delta M"] & & & S',\, M' \arrow[d, dash] & \\
D \arrow[r, dash] & {} \arrow[r] & I \arrow[r, dashed, "\text{SMOs}"] & I' \arrow[r, dash] & {} \arrow[r] & \Delta D
\end{tikzcd}$$

Figure 4.1: A schema of the workflow. First, the user defines the SMOs (red). Then the system decides if the update statements can be generated directly from the changes in mappings (blue, ΔM). If not, it has to create the instance category, apply the SMOs to it and then export the statements from it.

There are, of course, many other opportunities for optimizations. For example, in the third step, we can locate the affected parts of the schema and load only them to the instance category. However, as mentioned above, the framework is meant to be a proof concept, so we leave the possible optimizations for future work.

5. Supported Operations

In this chapter, we describe the particular SMOs we support.

We divide the operations on the conceptual layer into three groups – basic, composite and complex. Operations from the first group are simple, low-level, and involve only one object or morphism at a time. They may affect multiple objects or morphisms (end even use other basic operations to do so) – for example, deleting an object would result in deleting all its morphisms. However, all the affected entities can be automatically determined by the system.

Another essential feature of the basic operations is that they may both originate from and result in an invalid schema category. The general rule is that the last basic operation we use must transform the schema category to a valid state. Otherwise, the user will not be allowed to apply any of them.

On the other hand, both the composite and the complex operations transform the schema from one valid state to another. They involve multiple objects or morphisms and usually require more parameters than the basic ones, too. Although many of them can be emulated by several basic operations, it is not a universal rule. For example, the `SPLIT/JOIN` operation requires a custom user input which none of the basic ones can provide.

After that, we introduce the operations that affect only the logical models. Some of them are already implemented, which we will discuss in more detail later.

5.1 Basic Operations

The schema category is composed of two types of entities – the objects and the morphisms. The objects consist of the *label* and one or more *id*-s (elements of the *ids* property). There is also the *key*, i.e., the internal identifier, but it is not accessible by the user. The morphisms have the domain and codomain objects, the cardinality and the *signature*, which is again an internal identifier. Lastly, the *superid* property of an object is basically just a sequence of morphisms.

This determines all of the basic operations. Both the objects and the morphisms can be created, deleted and edited. To edit an object means to edit either its *label* or its *ids* (the *key* is immutable and *superid* is determined by the *ids*). The edition of the *superid* includes deleting one of them or adding a new one.[1] To edit a morphism means to edit its cardinality – changing its domain or codomain object would be too complicated and not really useful, so it is not supported, and the *signature* is, similarly to the *key*, immutable.

An important concept of these operations is that the entities are defined by their identifiers, not by their other properties. So if we decide to delete an object and later create a new one with all of its properties identical, the result will be one `ADD` and one `DELETE` operation. The system will have no way of knowing we meant these objects to be identical, so the mapping associated with the old

[1] An *id* is defined by its value, i.e., by a set of morphisms, instead of an internal identifier. Therefore editing *id* can be simulated by deleting it and creating a new one.

object and all the corresponding data will be erased. Hence, the user will also be required to provide a mapping for the new object.

5.1.1 `ADD Object`

Operation `ADD` creates a schema object and adds it to the schema category. The object's *label* has to be specified by the user. Besides that, there is no need for any user input.

After the operation, the schema category will be invalid because the object will have empty *ids*. It will also not be a part of any mapping because the system does not have the respective information. So this operation does not affect the logical models.

5.1.2 `DELETE Object`

This operation deletes an object. All morphisms linked to it are deleted as well (see the `DELETE Morphism` operation). This might, in turn, indirectly modify the current mappings or invalidate the schema category.

5.1.3 `EDIT Object Label`

A *label* is just a human-readable description of the object. It can be easily changed without any potentially problematic side effects.

5.1.4 `ADD Id`

The user selects an object and a set of morphisms that originate in it. The set (i.e., the new *id*) is then added to the object's *ids*. After the operation, the schema category might become valid. The current mappings are not affected in any way.

There are different rules depending on the type of the newly created *id*:

- *empty*, i.e., the object is a simple value property identified by its value – the set has to contain exactly the identity morphism,
- *simple* – the set has to contain exactly one non-identity morphism with cardinality (`?..1` $\leftrightarrow$ `1..1`),
- *complex* – the set has to contain multiple non-identity morphisms with cardinalities (`?..*` $\leftrightarrow$ `1..1`).

The second rule ensures that the object the morphism refers to is unique. The third rule prevents any part of the *id* from being unique enough that it would be able to create a simple *id* on its own.

5.1.5 `DELETE Id`

Deletes the chosen *id*, i.e., removes the set of *signatures* from *ids* (the corresponding morphisms are not affected). However, if it was the last *id* of its object's *ids*, the schema category would be invalidated.

5.1.6 `ADD Morphism`

This operation creates a schema morphism and adds it to the schema category. The user selects its cardinality from a predetermined set of options. The reason for this is that some cardinalities are not compatible with each other, i.e., between any two objects A and B can be more than one morphism (e.g., $f\colon A \to B$, $g\colon A \to B$, etc.) only if they satisfy certain conditions (see Appendix A for full explanation).

If the newly created morphism completes a cycle, multiple tuples of composite morphisms will be created between all possible pairs of objects that form the cycle. So the user is allowed to select only from cardinalities that will not result in an incompatibility within any of these tuples.

Creating a morphism neither invalidates the schema category nor affects the mappings.

5.1.7 `DELETE Morphism`

First, the system deletes the morphism and all *id*-s that use it (via the `DELETE Id` operation). Then, the morphism is removed from all mappings. Because each access path is a tree, the system removes any subtree that is connected by the morphism. So this operation might both invalidate the schema category and change some mappings.

5.1.8 `EDIT Cardinality`

The cardinality of a morphism bears crucial information about the relation between the domain and codomain objects of the morphism. Some changes between cardinalities are straightforward, e.g., (`1..1` $\leftrightarrow$ `1..1`) to (`1..1` $\leftrightarrow$ `1..*`), because the second one is automatically valid if the first one is valid. We denote them as trivial. On the other hand, an opposite process is impossible without either a loss of information (removing some elements from $I(B)$) or an introduction of redundancy (adding some elements to $I(A)$). These are called non-trivial.

There are 16 possible cardinalities with 240 transitions in total. However, it does not make much sense to change, e.g., cardinality (`1..1` $\leftrightarrow$ `0..*`) directly to (`0..*` $\leftrightarrow$ `1..1`), so we limit ourselves to the transitions where only one of the four digits changes. This gives us 64 transitions, 32 of which are unique (the other 32 are just mirror images of them, i.e., we can get them by substituting A for B and vice versa).

In Figure 5.1 we have depicted all the 32 unique transitions between the cardinalities. The derivation and full explanation of the diagram are depicted in Appendix A. The non-trivial transformations require a change of data. It is expected to be either removing some elements from $I(A)$ or $I(B)$ (marked as $\mathbf{A}^-$ or $\mathbf{B}^-$), adding them ($\mathbf{A}^+$ or $\mathbf{B}^+$) or removing some *active mapping rows* between them ($\mathbf{m}^-$).

Some of these changes can be automated (for instance, removing all $a \in I(A)$ elements that do not have any linked $b \in I(B)$ during the (`1..1` $\leftrightarrow$ `0..1`) to (`1..1` $\leftrightarrow$ `1..1`) transition) while the others need user input. It is expected to be

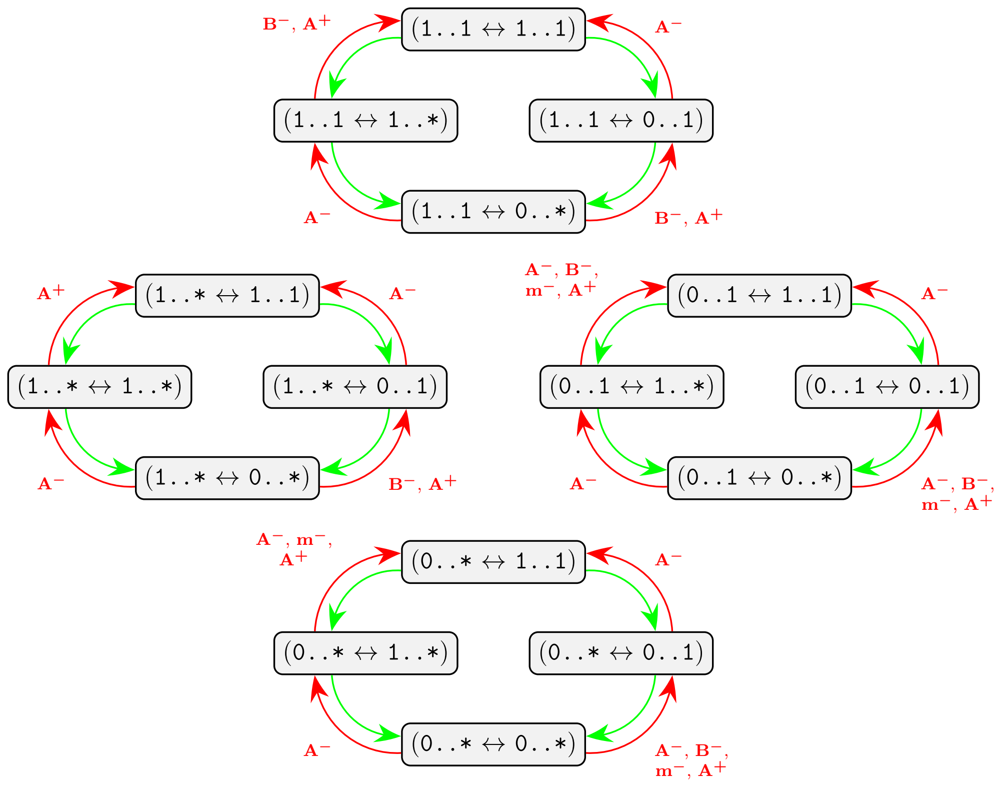


Figure 5.1: All possible cardinalities and both trivial (green) and non-trivial (red) transitions between them. For better clarity, half of the possible transitions are missing. However, they all are just mirror images of those depicted here. See Appendix A for more details.

a function $F\colon I(A)* \to I(A)$ that gets an array of elements[2] and chooses which one to keep.

If the morphism no longer conforms to the cardinality requirements described in Subsection 5.1.4, we have to delete all affected *ids*. In that case, the user will be warned before the operation, and if he or she chooses to continue, the system will proceed with the `DELETE Id` operations.

5.2 Composite Operations

These operations bring some quality-of-life improvements over the basic ones. The goal is to provide operations at the level the user thinks about the data models. For example, the user almost never needs just to create an object. He or she usually wants to also connect the object to another one, most likely with the intention of adding a new property to the existing object. Other examples include more sophisticated structures like maps, lists, etc.

All composite operations can be created by combining several basic ones. This fact is essential to ensure they are formally correct. Nevertheless, this does not mean they have to be implemented this way. Generally, the more complex an operation is, the more opportunities for optimization there are.

[2]The expression $I(A)*$ is our notation for an array of elements from $I(A)$, i.e., $I(A)* = [a_1, a_2, \ldots, a_n]$, where $a_1, a_2, \ldots, a_n \in I(A)$.

Another added value of these operations lies in the automatic propagation into the logical models. For example, let us consider there is a schema object that is also a part of a mapping. If the user adds a child property to it, the system can easily update the mapping accordingly. Or, imagine the user wants to add a list. The system can check if the database supports lists first. If not, it warns the user about the potentially unnecessary operation.

5.2.1 `ADD Property`

This operation creates a schema object and a morphism that connects it to another object specified by the user. He or she also selects *ids* of the newly created object (or leaves it empty). The operation can be nested in another one of the same kind (so the new property can use one of its child properties in the *ids*).

5.2.2 `ADD Reference`

This operation creates a morphism with cardinality (`1..1` $\leftrightarrow$ `1..1`) between two objects (denoted as A and B) and adds a selected identifier of A to B. This means that B is now identified by one of the identifiers of A.

Optionally, the user can choose not to specify B. If that is the case, a new object is created first.

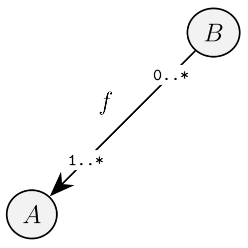
5.2.3 `ADD Relationship`

This operation creates a link between two objects (A and B) that correspond to a relationship from a relational database. There are two options:

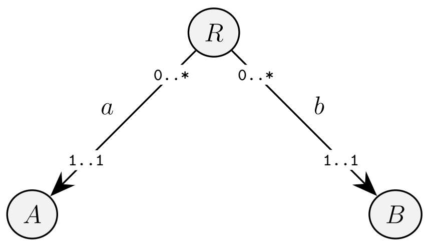
- One-to-many – a morphism $f\colon B \to A$ with cardinality (`0..*` $\leftrightarrow$ `1..1`) is created. A selected *id* of A is added to *ids* of B. The structure is depicted in Figure 5.2(a).
- Many-to-many – an object R is created and connected to A and B by morphisms $a\colon R \to A$ and $b\colon R \to B$. Both of them have cardinality (`0..*` $\leftrightarrow$ `1..1`). Object R is identified by an identifier from A combined with another from B. See Figure 5.2(b).

5.2.4 `ADD Set`

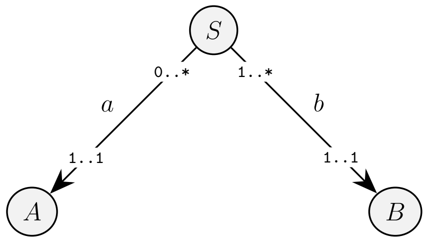
This operation creates an object B that acts as a set of properties for the selected object A. The final structure, depicted in Figure 5.2(c), is similar to the many-to-many case of the `ADD Relationship` operation (except that the b morphism has cardinality (`1..*` $\leftrightarrow$ `1..1`)). Although some database systems support arrays, and it would be enough to create only morphism $f\colon B \to A$ with cardinality (`0..*` $\leftrightarrow$ `1..*`) for them, we have chosen the more general approach because it works for all databases. Also, the morphism f is contained in the structure in the form of $f = a \circ \overline{b}$ so the mentioned databases can just use it while ignoring S.



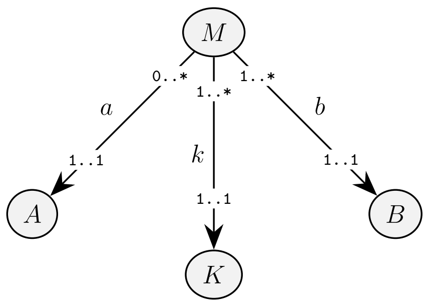
(a) Two schema objects connected with a `1:n` relationship.



(b) Two schema objects connected with a `n:m` relationship.



(c) Set of properties B associated with kind A.



(d) Map of keys K to properties B associated with kind A.

Figure 5.2: Various data structures that can be created by the composite operations.

5.2.5 `ADD Map`

Unlike the previous operation, Map has to be in all types of databases represented in the same way. It is a relationship with an additional object that acts as a key. So for selected object A, this operation creates objects B, M, K and morphisms $a\colon M \to A$, $b\colon M \to B$, $k\colon M \to K$. The cardinality of a is (`0..*` $\leftrightarrow$ `1..1`), the other two have (`1..*` $\leftrightarrow$ `1..1`). The object M is identified by the combination of A and K. The final structure can be found in Figure 5.2(d).

5.2.6 Notable Exclusions

All the previous operations have in common that they only create new structures and add them to the schema category and the corresponding mappings. However, there are none that would delete or edit already existing structures.

The `DELETE` ones are excluded because they are already covered by the `DELETE Object` operation (which subsequently calls `DELETE Morphism` and so on). Whenever we want to create a structure, we have to specify it in detail. So it is convenient to have available all the different `ADD` operations for all the expected structures. On the other hand, deletion is much simpler. The user can just select all the schema objects and delete them at once.

The `EDIT` operations are missing for the opposite reason. To update an object means to update all data of the corresponding instance object. These potentially really complex operations are therefore left until the next section. The only trivial one, i.e., renaming, is covered by the `EDIT Object Label` operation.

5.3 Complex Operations

Unlike the operations from the previous section, which specialized in creating and deleting data structures, the complex ones aim to mediate transformations on already existing parts of the schema category. This means they also modify the data of the corresponding instance objects and morphisms.

5.3.1 `COPY`

This operation aims to create an object with the same properties and data as another object and then move it to a different place in the schema category graph (i.e., link it to a different object than the original one is linked to). During it, a change of cardinality might be needed. Although that looks like an easy task, there are many hidden issues.

First Iteration

Let us first consider the most basic yet non-trivial example, depicted in Figure 5.3. There are three objects:

- `user` – each user account is created simultaneously with his or her first order. However, the user can then make any number of additional orders (morphism b).
- `order` – each order is created by exactly one user.
- `contact address` – a string representing the user's contact address (morphism a). It is also used as a delivery address for the user's orders.

Although this schema might look fine for a simple e-shop, there is a slight inconvenience. The user can not customize the delivery address for each order and has to use the default one instead. So we want to edit the schema by adding a new object, called `delivery address`, which will be linked directly to the order. We also want to ensure that all currently existing orders have valid addresses, so we want to copy them from the users.

Note the `contact address` has (`1..1` $\leftrightarrow$ `1..*`) cardinality to the `order` while the new `delivery address` should have (`1..1` $\leftrightarrow$ `1..1`). Consequently, a change of cardinality is required. We can do it as follows:

1. We create a new object, `delivery address`, that is a copy of `contact address`.
2. We create a morphism f: `order` $\to$ `delivery address` by combining the currently present morphisms a and b to $\overline{(b \circ a)}$ and replacing `contact address` for `delivery address`. Its cardinality is (`1..*` $\leftrightarrow$ `1..1`).
3. We transform f to f' by changing its cardinality to (`1..1` $\leftrightarrow$ `1..1`) by the $\mathbf{B}^+$ process.

We have ended up with redundant data in the form of duplicated addresses. Still, it is all right because, in the future, the delivery addresses will not be necessarily identical.

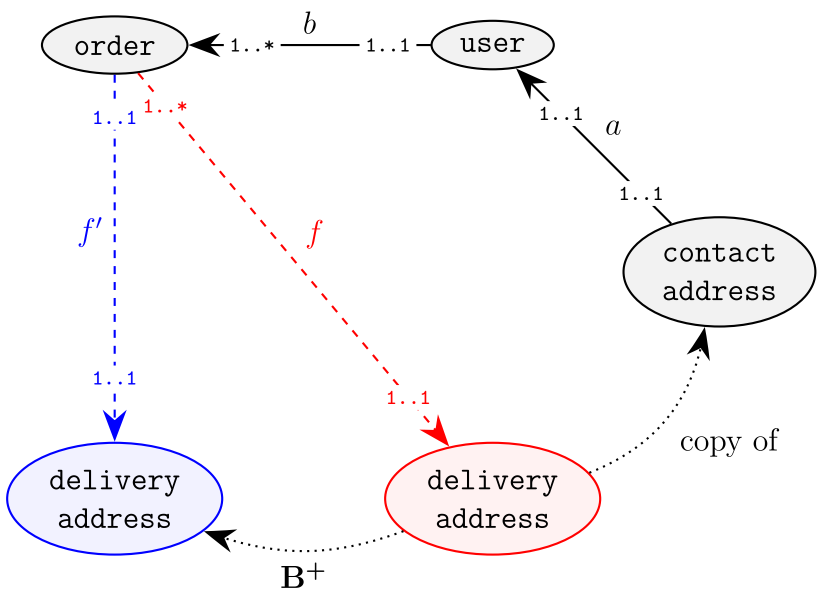


Figure 5.3: An example of the first version of the COPY operation. Object **contact address** is copied to **delivery address**.

Better Algorithm

However, this process has one crucial flaw – the object created in step one does not have any direct connection to the **user**. It can not even have because then step three could not happen. During it, a $(\texttt{1..1} \leftrightarrow \texttt{1..1})$ morphism (f') is created, and that one is not compatible with the already existing compound morphism $\overline{(b \circ a)}$. The issue is that the transformation might require additional information from the other objects. Let us show it in another example.

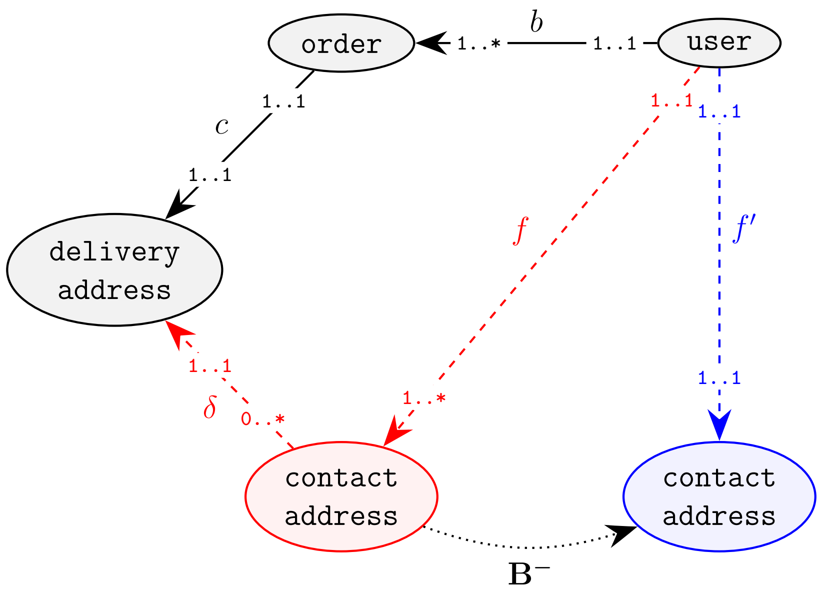


Figure 5.4: An example of the final version of the COPY operation. Object **delivery address** is copied to **contact address**.

Consider a very similar situation, depicted in Figure 5.4. Now each order has a unique address (morphism c) while the users have none. This is also a bad design because we may need a way to contact the users. So we have to create a `contact address` object. Naturally, it should be the last one the user used for delivery, so we just need to order the addresses by their orders' dates, pick the last one, and copy the value to the `delivery address`. But we need to maintain a connection to the order object since the date is stored there. We can solve it by this process:

1. We create a new object, `contact address`, that is a copy of `delivery address`. It is linked to the original `contact address` by a ($\texttt{1..1} \leftrightarrow \texttt{1..1}$) morphism $\delta\colon$ `contact address` $\to$ `delivery address`.

2. We create a morphism $f\colon$ `user` $\to$ `contact address` by combining the currently present morphisms b, c and δ, i.e., $f = \overline{\delta} \circ c \circ b$. Its cardinality is ($\texttt{1..1} \leftrightarrow \texttt{1..*}$).

3. We change the cardinality of δ to ($\texttt{0..*} \leftrightarrow \texttt{1..1}$).

4. We transform f to f' by changing its cardinality to ($\texttt{1..1} \leftrightarrow \texttt{1..1}$) by the $\mathbf{B}^-$ process.

5. We delete the δ morphism.

Note that during the transition in step four, we still have full access to the original `delivery address` object and all its linked objects thanks to the `1..1` end of the δ morphism. So we can easily select the latest contact address for each user. The change from `1..1` to `0..*` in step two is necessary to ensure that both f and f' can coexist along the $\overline{\delta} \circ c \circ b$ morphism. Because δ is of the fourth cardinality type, it is compatible with any other morphism, so this choice of δ is sufficient for all cases.

More Complex Objects

We managed to decompose the `copy` operation for a simple value object as a sequence of basic operations. The next step is to consider a complex object, an example of which is shown in Figure 5.5. The `address` has three properties:

- `id` – a unique identifier of the address,
- `value` – a string with the city, street and house number.
- `note` – a set of string notes associated with the address, e.g., "Beware of the dog".

Granted, it is not the most efficient address design, but it is just an example. The question is what to do if we decide to copy the object. Generally, there are two approaches to copying objects – a shallow or a deep copy. We can do something in between by deciding individually for each property if we want to copy it as well or just link it to the new object. The choice depends on its semantics:

- `note` – this property is specific to each address, so we will copy it.

- `value` – multiple addresses share this property, so we probably should not copy it. However, it is unclear if it can be shared not only by different addresses but also by different objects, i.e., `contact address` and `delivery address`. If that were not the case, we would have to copy it.

- `id` – this property is obviously meant to be unique for each address. However, copying it might not be enough. If we needed to add redundancy, the uniqueness would be lost.

The example illustrates that there is no general solution. Instead, the user has to choose which properties to copy manually. This choice has to be made between the fourth and fifth steps in the above-described algorithm. There is still present the auxiliary morphism δ, which can be used as a link to the original object. Then we repeat the algorithm for each property the user wants to copy (and subsequently for their subproperties, if there are any, and so on). Finally, we can delete all the auxiliary morphisms.

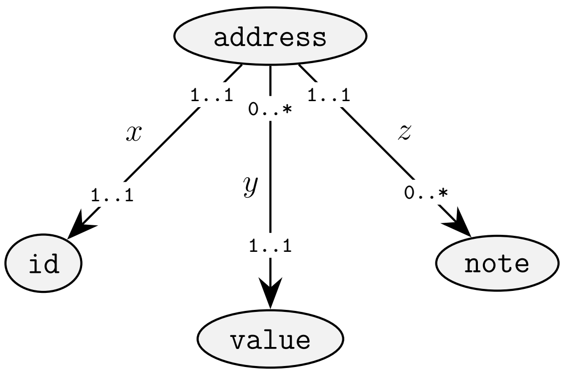


Figure 5.5: An example of a complex object `address`.

We also have to provide a generator of unique identifiers so that new values for the `id` can be created. The user might require a particular format for these values (e.g., some identifiers are numbers, others are strings of letters, ...). If that is the case, he or she can input a custom generator function.

Lastly, we have to fix *id*s that might have been invalidated by the operation, i.e., those that use any morphism originating in the moved object. Although some of them may be repairable automatically, it generally is not the case. For example, if `delivery address` was identified by its `order`, the user would have to specify a new identifier (e.g., `user`).

The most simple solution is to invalidate all the affected *id*s and then let the user fix them manually. The development of an algorithm that would allow us to automatically repair at least some is left to future work.

5.3.2 `MOVE`

This operation does exactly the same as the previous one with one important difference. The original object is deleted after the copy is made. The process can be optimized by keeping the original object and transforming the morphisms. However, this is just a question of the implementation of the algorithm. From the operational point of view, it is much better to take the already proven concept and extend it a bit.

5.3.3 `GROUP`

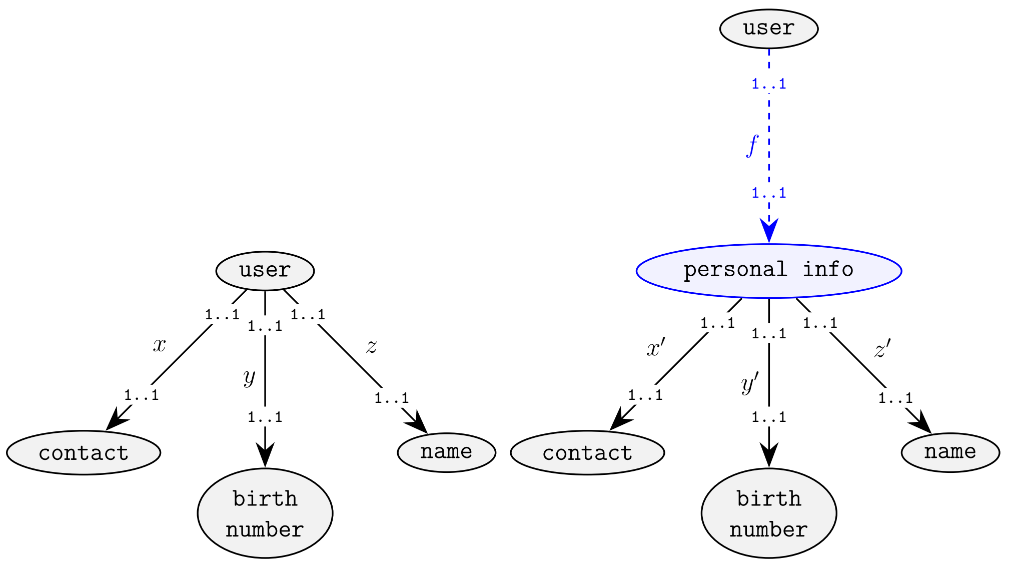
This operation replaces multiple properties with a new object that has them as its subproperties. The most basic cases can be done by creating the new object and then moving the other properties to it. However, the more advanced ones might require additional transformations.

One to One

Consider the situation depicted in Figure 5.6(a). The `user` object has several properties (`contact`, `birth number`, `name`) representing the user's personal information. We want to ensure this information is well protected, so we decide to store it together, possibly in another database system. In order to do so, we have to create a new object, `personal info`, and move all the properties to it. We also can not edit the `user` object, because it might be linked with other properties (e.g., `id`, `orders`, ...).

The result is shown in Figure 5.6(b). It is important to note that the `personal info` has to be identified by the `user` so that the f morphism can have the (`1..1` $\leftrightarrow$ `1..1`) cardinality in all possible cases.

We could have used the `ADD Object` and `MOVE` operations instead. However, the significance of the `GROUP` operation is that it can internally optimize all the data transformations. And it also should be much more user-friendly.



(a) Before. (b) After.

Figure 5.6: An example of the `GROUP` operation with `1:1` cardinality between the kind and its properties.

One to Zero

This example illustrates other benefits of this approach. As we can see in Figure 5.7(a), there is, again, a `user` object with some properties (`street`, `city`, `ZIP`). The last two are optional because we were not specific enough, and some users did not fill in the corresponding fields in a form. All the properties can also be linked to multiple users as there might be multiple people from the same city (or street or ZIP zone).

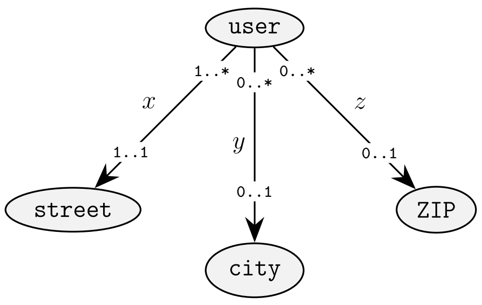
We want to group the properties to one `address` object. Furthermore, we would like to fix some of the previous mistakes by eliminating all non-valid addresses, i.e., those which miss at least one of the three properties. In other words, we want the whole address to be optional instead of its properties, as shown in Figure 5.7(b).

This issue can be solved entirely by the previously mentioned operations. First, we do the same `ADD Object` and `MOVE` procedure as in the example from the One to One case. This creates the ($\texttt{1..1} \leftrightarrow \texttt{1..1}$) morphism f while leaving the x, y and z ones intact. Then we change f to ($\texttt{1..1} \leftrightarrow \texttt{1..0}$). Lastly, we use the $\mathbf{A}^-$ transformation on y and subsequently on z, meaning we delete the addresses without a `city` or a `ZIP`.

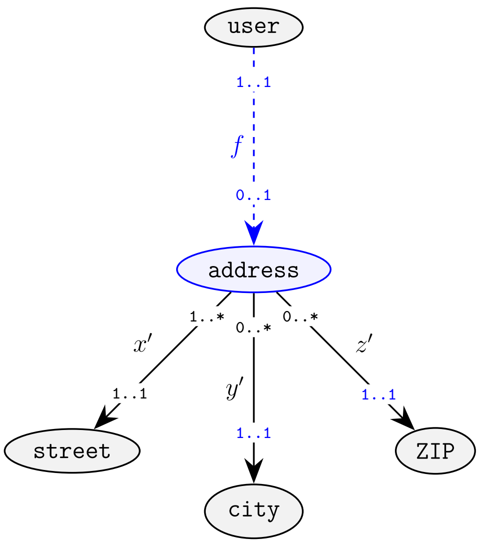
Again, the purpose of the `GROUP` operation is to streamline the transformation as much as possible. It should also be general enough to allow us to choose a different rule for creating addresses. We can simply ask the user for a function like

$$G\colon I(\texttt{street}), I(\texttt{city})?, I(\texttt{ZIP})? \to \{\texttt{true}, \texttt{false}\}\,, \tag{5.1}$$

where $I(A)?$ is our notation for an optional input of an element from $I(A)$, i.e., $I(A)? = \{a, \emptyset\}$ where $a \in I(A)$. The function takes the values of the objects that would be used to create the address and decides if to create it or not.



(a) Before.



(b) After.

Figure 5.7: An example of the `GROUP` operation with `n:1` cardinality between the kind and its properties.

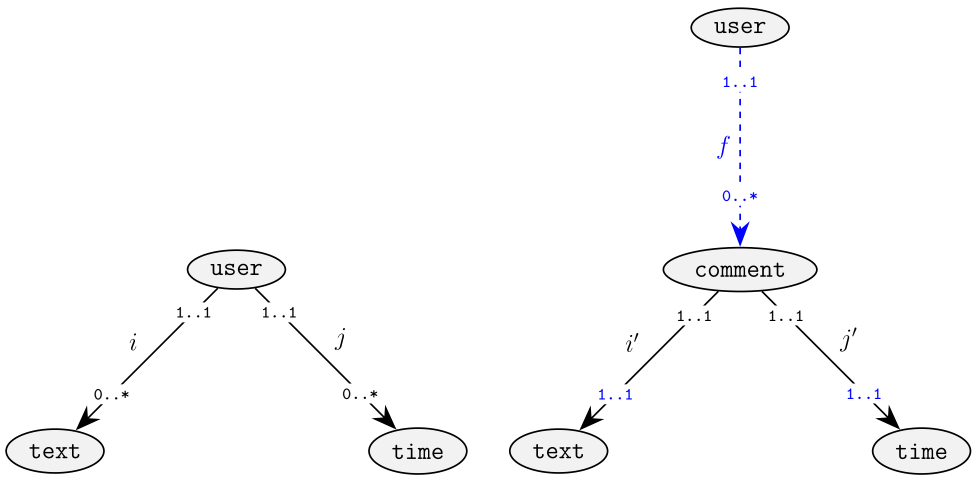
One to Many

Even more complex example is depicted in Figure 5.8(a). There is a `user` with only two properties, arrays `text` and `time`. They represent comments the user has made – each of them has its content in the `text` array and time of publication in the `time` array. This design was probably sufficient at the beginning when we only needed to read the comments all at once. Later, we decided to implement more sophisticated features, i.e., querying individual comments by time or content. So we would like to transform the schema to the one in Figure 5.8(b).

Unlike in the previous examples, this problem can not be solved by the basic operations only. We have no choice but to use the `GROUP` operation. The function (5.1) can be generalized to

$$G\colon I(\texttt{text})*, I(\texttt{time})* \to I(\texttt{comment})*\,, \tag{5.2}$$

where $m* \subset I(\texttt{text})*$ and $n* \subset I(\texttt{time})*$ are the arrays of the instance elements corresponding to the given user element and $o* \subset I(\texttt{comment})*$ is the output array of comments (tuples of type $(m*, n*)$). Of course, there can be multiple such arguments as $m*$ and $n*$, this is just an example. The important part is that because they and the output are arrays, this approach covers all possible cardinalities of the morphisms between the central object and its attributes. Moreover, the (m, n) tuples would be sufficient in this case, but not generally since the cardinalities of the i' and j' morhpisms might be arbitrary.



(a) Before. (b) After.

Figure 5.8: An example of the `GROUP` operation with `1:n` cardinality between the kind and its properties.

Finally, we can generalize the function (5.2) even more. The object `user` is linked through various morphisms to all specific values of its properties. Therefore we can write

$$G\colon I(U) \to I(O)*\,, \tag{5.3}$$

where $I(U)$ is the instance of the central object (i.e., `user`). So the `GROUP` operation can be understood as a transformation that takes:

- an object U,

- a set of objects $\{A_i\}$, where each element is linked to the U by a base morhism $x_i \colon U \to A_i$ with cardinality $(\overline{c_i} \leftrightarrow c_i)$,
- a set of output cardinalities $\{c'_i\}$,
- an output cardinality c',
- a function of type (5.3).

Then it:

- creates an object O and a morphism $f \colon U \to O$ with cardinality $(\texttt{1..1} \leftrightarrow c')$,
- links each A_i with O by a new morphism $x'_i \colon O \to A_i$ with cardinality $(\overline{c_i} \leftrightarrow c'_i)$,
- for each $u \in I(U)$ computes the $o* = (a_1*, a_2*, \dots)* = G(u)$ elements, where $a_i \in I(A_i)$, inputs the o to $I(O)$ and links it with u and a_i* elements by the f and x'_i morphisms,
- deletes the original x_i morphisms.

It is important to remember that the choice of the function G depends on the selected cardinalities c' and c'_i. For instance, if $c' = \texttt{1..1}$, then G has to be from $I(U)$ to $I(O)$ instead of $I(O)*$. Or $c' = \texttt{0..1}$ means $G \colon I(U) \to I(O)?$. On the other hand, c'_i affects the cardinality of a_i in the output tuple o. The arrays are used here because it is the most general notation (e.g., u is just an array with length 1 while $u?$ is an array with length ≤ 1) but for practical purposes (i.e., during implementation) it is necessary to check if the function submitted by the user meets the cardinality requirements.

The limitations on the function imposed during the implementation also impact the operation's effectiveness. If we know, for instance, that the submitted function uses only two properties of U, we do not have to input the whole u into it. We can only provide it with the values of these two properties instead. So we save a lot of time by not having to fetch all the unnecessary data. The definition of G in (5.3) shows merely what the `GROUP` is able to do in its most general sense. There is plenty of room for optimizations, but they are a matter of implementation.

5.3.4 `UNGROUP`

This is (in most cases) the inverse operation to `GROUP`. The exception is when `GROUP` is not reversible, i.e., data is lost in the process. Then `UNGROUP` acts only as a partial inversion.

An example of that is in the One to Zero case. The schema in its original form, after `GROUP` and then after `UNGROUP` is depicted in Figures 5.7(a), 5.7(b) and 5.9. Because we wanted to create the address only when all its three components were present, their values in all other cases were lost. Later, when we decided to transform the schema back, the lost data could not be restored. This is also illustrated by the fact that the x'' morphism now has cardinality $(\texttt{1..*} \leftrightarrow \texttt{0..1})$ instead of $(\texttt{1..*} \leftrightarrow \texttt{1..1})$ that its original version x had.

The operation can be defined as follows. It starts with:

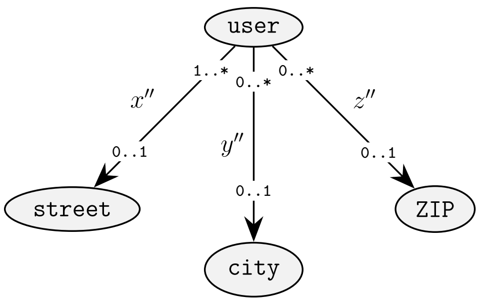


Figure 5.9: An example of the `UNGROUP` operation following on from the previously executed `GROUP` operation (depicted in Figure 5.7).

- objects U and O, linked by a morphism $f\colon U \to O$,
- a set of objects $\{A_i\}$, linked to O by the morphisms $x'_i\colon O \to A_i$,
- no other objects than U and $\{A_i\}$ are linked to O.

Then it:

- links each A_i with U by a new morphism $x''_i = x'_i \circ f$,
- deletes the O object and all its morphisms (i.e., f and all x'_i).

Note that the operation is much more straightforward than `GROUP` and does not require any user input. It also does not lose any data on its own – it is only unable to restore the data lost by a previous application of `GROUP`, although it might cause a loss of information. For instance, if we used it to reverse the schema depicted in Figure 5.8(b) to its original form as is in Figure 5.8(a), we would no longer be able to tell which `text` belongs to which `time`.

Moreover, `UNGROUP` does not have to be used merely as a (partial) inversion of `GROUP` because the schema it is applied to does not have to fulfill the cardinality restrictions of the output of `GROUP` (i.e., the (`1..1` $\leftrightarrow$ `?..?`) of the f morphism). To sum it up, this operation is useful for removing a specific object from the middle of the schema category while preserving the relations between all others.

5.3.5 JOIN/SPLIT

All the previous operations affect only the links between the objects but never their values. They can delete the data but are not able to create new ones. This operation aims to do exactly that.

The `JOIN` is a well-known function in many programming languages. It is used to join multiple string arguments (or an array) by a string separator. On the other hand, `SPLIT` does exactly the opposite – it takes one string, finds all occurrences of a separator and splits it by them.

JOIN

Let us consider an example depicted in Figure 5.10. We have a `user` object whose name is divided into two properties, i.e., `first name` and `last name`. This is not the best design possible because a person can have more than two names. When the first such people start to register into our application, we decide to create one universal object called `full name`. We have to migrate all current users, and in order to do so, we have to calculate their full names by joining the first ones with the last ones.

However, sometimes joining the names with a separator is not enough. Imagine that our application also considers the academic degrees of its users, which are stored as an array in the `degree` object. And we want to include them in the `full name` as well. The problem is they are separated by a comma and a space, i.e., the `", "` symbol, while names use just the space, i.e., `" "`. Although we could join the names first with the spaces and then add the degrees with the commas, it would still be an incomplete solution. Some degrees should be placed before the name and the other after it. They also should be listed in a particular order.

As we can see from the example, a custom transformation function is needed. In this case, it would look like

$$J\colon I(\texttt{first name}), I(\texttt{last name}), I(\texttt{degree})* \to I(\texttt{full name})\,. \qquad (5.4)$$

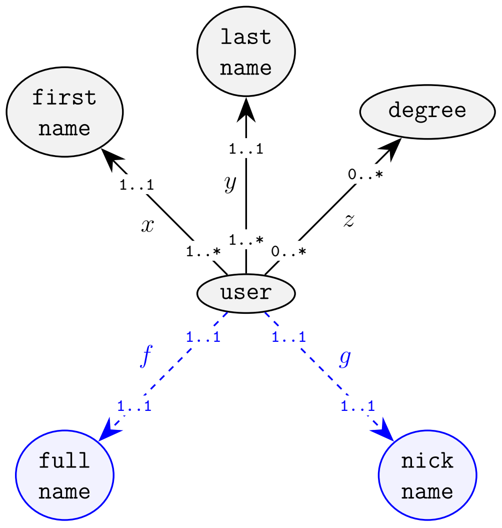


Figure 5.10: An example of the `JOIN/SPLIT` operation. The objects `full name` and `nick name` are created and their values are computed from the values of the other properties of `user`.

SPLIT

The `SPLIT` operation can be understood in a much more general sense, as well as `JOIN` is. We can provide more than one separator or even a regexp. The next step is to, again, provide a custom function. Unlike `JOIN`, the `SPLIT` function should take one argument and produce multiple output values.

All in all, both operations are almost the same. Both select one (or more) properties of an object and transform them into multiple (or one) new ones. The only distinction is in the domain and codomain of the user-provided transformation function. However, that is not a significant difference, as we have seen in the One to Many case. So we can generalize both operations even more by combining them into one that accepts any number of properties and outputs any number of new ones.

An operation like this has many use cases. Consider, for instance, the previous example with user names. We might want to transform the three properties (`first name`, `last name` and `degree`) into two new ones – the already mentioned `full name` and a `nick name`, which is supposed to be a much shorter version for informal usage. We might start by discarding the degrees and abbreviating the last name.

Lastly, we can generalize even further by allowing the transformation function to accept the whole $u \in I(\texttt{user})$ element so it can access all its related objects. There is actually no need to limit them to only those that are linked to u with a base morphism. So the function can be defined as

$$J\colon I(U) \to I(O_1) * \times I(O_2) * \times \cdots \times I(O_n) * , \tag{5.5}$$

where U is the central object and O_i are all the new objects created in the process. The user has to select U, input J and specify all the O_i objects. The `JOIN/SPLIT` operation then do the following:

- it creates all output objects O_i and links them with U,
- for each $u \in I(U)$, it computes the value of $J(u)$ and adds its components to the corresponding $I(O_i)$.

Note the definition does not include the source objects (i.e., whose data are used to calculate the new values) because they can all be accessed from u. However, in practice, the user will have to specify them first in order to be able to use them in the J function. That is, like during the `GROUP` operation, crucial for the efficiency of `JOIN/SPLIT`.

The operation's name might suggest that the source objects should be deleted after. We suggest leaving this behavior optional (and ideally individual for each object) because there are many use cases for both possibilities. In any case, the objects can be deleted independently after the operation, so it is just a matter of optimization.

5.4 Operations in the Logical Layer

Lastly, we describe how the user can change the logical models. Because the models are defined via the mappings, the only way to change a model is to edit its respective mapping. And a mapping is, in turn, characterized by its access path, which is a tree of properties. So most of the options available to the user are tree operations.

Unlike the conceptual model, the logical models are platform-dependent. This manifests itself in the form of limitations any given access path must conform to. In other words, each database system allows a slightly different set of structures the access path can contain. Therefore, it is impossible to define them as strictly and generally as those acting on the conceptual schema. For each database, they can differ in complexity, possibilities, or, in some cases, they might not be allowed at all. Such examples are listed with the specific operations below.

As a reminder, these operations can not propagate to the conceptual schema. To change it, an operation that originates there, i.e., one of the previously mentioned, is required instead. The mappings are also not related to each other, only to the conceptual schema. So editing the schema category can affect all of them, but any operation on a specific mapping does not propagate to any other. This greatly reduces possible problematic side effects of the operations.

5.4.1 Migration

This operation is rather an abstract concept overreaching the other ones instead of a specific set of well-defined steps. However, it is incredibly useful because it solves one of the issues of multi-model databases – transferring the data from one database system (**A**) to another (**B**). In order to do so, the user has to first define the mappings from **A** and **B** to the conceptual schema and then run the transformation algorithms.

5.4.2 Property Level

There are two types of nodes in the access path – simple and complex. The first ones are leaves with a simple value (e.g., string). The second ones represent the inner nodes of the tree. They do not have simple values, but they can have multiple child properties. Some databases (e.g., most of the relational ones) do not support nested properties, so they have only one complex property, i.e., the root of the access path. Further, additional rules might be in place. For instance, MongoDB requires each kind to have the `_id` property.

With that in mind, we can construct the basic operations:

- `ADD` – creates a new property and adds it to a specified position in the tree.
- `DELETE` – deletes a property. If it was a complex one, all its children are deleted as well.
- `RENAME` – changes the string identifier associated with the property.
- `MOVE`, `COPY` – moves / copies the property to another position in the graph.

In addition, any number of complex, mostly platform-specific operations can be implemented to make the tool more user-friendly. However, we should be able to define them as a sequence of the basic ones, so they do not bring anything particularly new.

What is more interesting are the ways the operations can be processed. For example, `RENAME` in a relational or most columnar databases can be done by one statement that simply renames the corresponding column. But in a document database, we might need to execute the operation on all data entities separately. So there are a lot of opportunities for optimization. Overall, the general approach is to import all data to the instance category, do the transformations, and then export them back. However, we can implement platform-specific algorithms that would search for these optimizations and use them whenever possible.

5.4.3 Kind Level

The options we have on the whole mappings are rather limited. We can `CREATE` a new one and `DELETE` or `RENAME` an old one. To move a mapping does not make sense, however, we can `COPY` it. Unlike any other above-mentioned concept (e.g., schema objects, access path properties, etc.), multiple identical mappings might exist at the same time. We can still differentiate them by an internal identifier, but their content, i.e., the access path and database, can be the same.

All the operations are completely safe. The only potentially problematic situation is when the user creates two or more mappings that share both the kind's name and the database, which can lead to an attempt to create a table (or a collection etc.) that already exists. Although it is up to the user to ensure that both mappings are not used to export data simultaneously, we can at least warn the user if this situation shall happen.

An interesting and potentially useful operation would be to copy a mapping while automatically changing its database system. There are still plenty of unresolved issues, but they are probably not that challenging, considering that all access paths generally have a similar structure. If both databases had the same restrictions, the access path would not have to change at all. If that was not the case, a transformation would be needed. However, the development of such an algorithm is not in the scope of this thesis, so we leave it as an open challenge.

6. Implementation

In paper [9] the authors proposed the usage of category theory as a general enough approach to represent multi-model data. They also defined all necessary concepts, derived algorithms for data transformation, and described the idea of a framework capable of storing multi-model data, transforming them via these algorithms and ultimately running queries on them.

We implemented these concepts in the form of MM-cat framework within the research project *A Tool for Modelling of Multi-Model Data*. We wrote article [10] that is currently under review at an international conference.[1] MM-evocat, the extension of MM-cat that focuses on evolution management, is also described in the article. Its implementation is mostly left to future work.

The codebase is hosted on GitLab[2]. The running demo version can be found here[3]. There is also a more detailed documentation.[4]

6.1 Architecture

First, note that the current implementation is meant to be purely a proof of concept. Although the computational requirements were taken into account when deciding what algorithms to use and how to implement them, they were never considered to be a priority. The primary purpose of the application is to show that the chosen approach for evolution management of multi-model databases is viable. The optimizations will be done as a part of future work.

For the same reasons, the application does not use any security policy, meaning anybody can use it without authorization. The only security mechanism is that the backend application does not expose passwords of the database accounts it uses to import data from.

6.1.1 Modules

The overall architecture is depicted in Figure 6.1. The framework is divided into several modules.

`core`

This module defines the basic objects and structures the rest of the framework relies on. It is represented by the yellow boxes, but it is used in all other modules.

`transformations`

The two blue boxes include algorithms, which form the central part of the module `transformations`. Their purpose is to transform data from the databases to the categorical representation and back.

[1]31th ACM International Conference on Information and Knowledge Management
[2]`https://gitlab.mff.cuni.cz/contosp/evolution-management`
[3]`http://nosql.ms.mff.cuni.cz/mmcat/`
[4]`http://nosql.ms.mff.cuni.cz/mmcat-docs/`

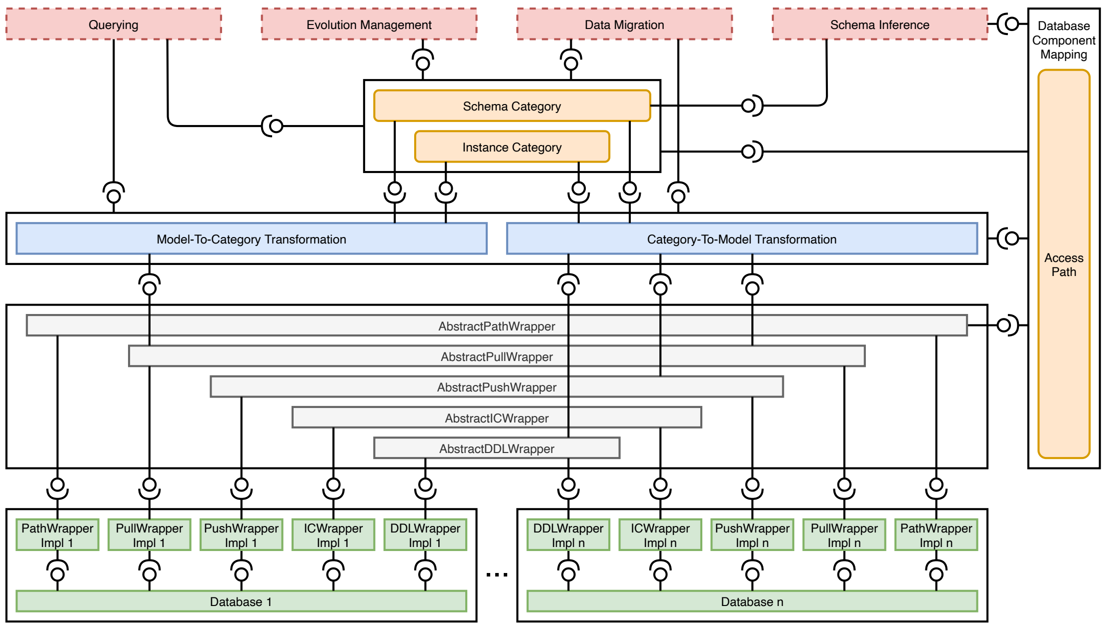


Figure 6.1: The architecture of the MM-cat framework.

`abstractWrappers`

Depicted in gray, these classes hide the specific databases and provide a platform-independent interface used by the algorithms from the `transformations` module.

`wrappers`

Each supported database system has its own wrappers which are a concrete implementation of the `abstractWrappers` module. These communicate directly with the drivers of the selected database. Currently, the following database systems are supported:

- *PostgreSQL*[5], version 12.11 – primarily a relational database with a support for the key/value and document (JSON, XML) models.
- *MongoDB*[6], version 4.4 – a document (JSON) database with support for the key/value model.

`server`, `example-ui`

The red boxes represent all high-level operations the user can do with the framework:

- querying over the categorical representation,
- inference of categorical schema from the actual data,
- migration of data from one database system to another,
- evolution management of the categorical representation.

[5]`https://www.postgresql.org/`
[6]`https://www.mongodb.com/`

Although both the querying and schema inference operations are beyond the scope of this thesis, they are closely related to it [11]. The inference would allow us to automatically propagate the changes in the data to the schema category, as in the Darwin framework.

The queries are partially contained in the `JOIN/SPLIT` operation – it uses them, albeit in a very limited form, to gather the data from which the new values are computed. Another exciting topic is the propagation of schema changes to the queries. However, this is already covered by another thesis in progress [4].

6.2 Backend Application

The backend is based on the Spring Boot framework[7]. Its purpose is to make the data structures and transformation algorithms available. The algorithms are executed in the form of jobs. They are processed asynchronously (independently of the other components of the server) but serially (only one at a time) because the operations are not designed to run concurrently.

6.2.1 Architecture

The application follows a Controller-Service-Repository pattern whose primary goal is to separate business, presentation and data-storage logic. Although this concept is similar to the widespread MVC pattern, there are a few differences.

Controller

A controller is responsible for providing a REST API for the client application. Whenever a request is received, its content is parsed and all actual work is delegated to the Service layer. Later, when the request is processed, the response is created, converted to JSON and sent back to the client.

There are several sections (schema category, mappings, jobs, etc.), so each has its own controller class. Most controllers have their own service classes to delegate the work to, and those, in turn, have their own repository classes.

Service

A service contains all business logic. It is used by a controller to handle requests. Whenever it needs to fetch or persist data, it delegates this task to a repository.

Repository

A repository communicates with the application's database and maps high-level concepts to specific database commands. Currently, we use PostgreSQL.

[7]`https://spring.io/projects/spring-boot`

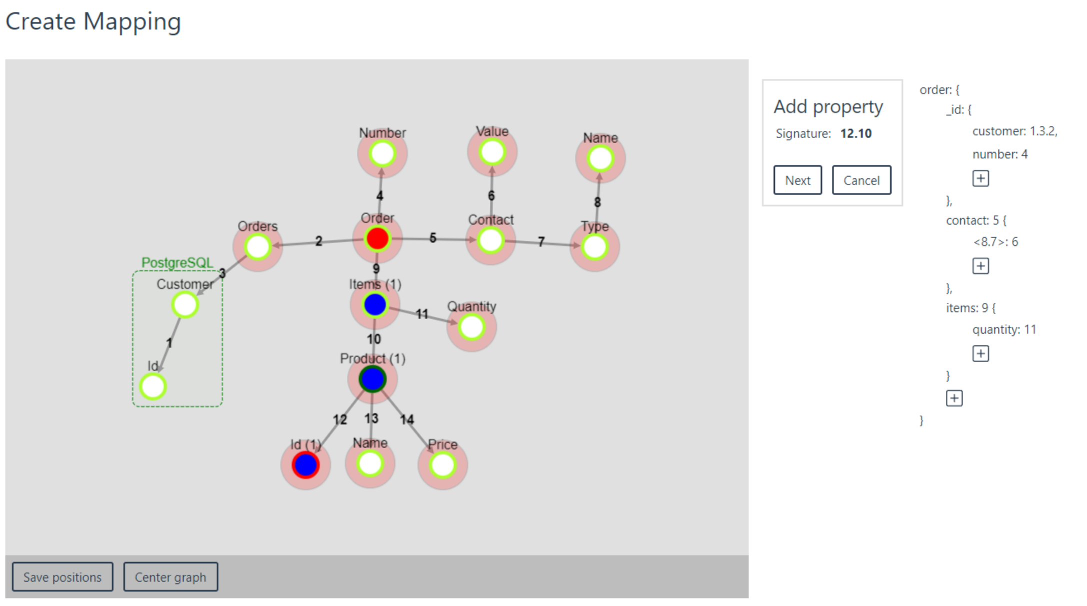
6.3 Frontend Application

The application is based on the Vue 3[8] framework with Vite[9] TypeScript and Options API. Its main purpose is to provide tools for defining and executing transformation jobs. Because of this, it is a single-page application with client-side rendering and an internal router.

The most important features of the frontend are the schema category and the mapping editors. The first one is used to create and edit a schema category. An example of that is depicted in Figures 7.1 and 7.3. It will be extended to enable all the conceptual SMOs defined in Chapter 5. The second one allows us to create mappings, as we can see in Figure 6.2. We plan to extend it to support the editation of already existing mappings.

6.3.1 Architecture

Vue is based on the Model-View-ViewModel pattern. The whole application is hierarchically divided into components, each of which is basically a tiny instance of an MVVM application.



Signature: **12.10**

```
order: {
    _id: {
        customer: 1.3.2,
        number: 4
        [+]
    },
    contact: 5 {
        <8.7>: 6
        [+]
    },
    items: 9 {
        quantity: 11
        [+]
    }
    [+]
}
```

Figure 6.2: Screenshot of the mapping editor of the MM-cat tool. We can see the creation of the mapping from the example in Figure 3.5.

MVVM

The model consists of data and business logic, and it is the source of truth of the application. The view-model (also called binder) controls what should be provided to the view. Whenever it receives an input from the view, it processes it and updates the model accordingly. The view presents the data to the user. It also captures user inputs and sends them to the view-model.

[8]`https://vuejs.org/`
[9]`https://vitejs.dev/`

Vue

The important part is that the framework does the binding between the view and the view-model. Vue automatically re-renders the corresponding part of the displayed view whenever a variable changes. Also, Vue sets the respective variable to the new value whenever a user changes an input.

This behavior is similar to other popular web frameworks, i.e., React.[10] The difference here is that while React uses immutable data structures and always re-renders the whole view, Vue reacts to the mutations of the data. Then it finds out what exactly changed and re-renders only what is necessary.

Components

A component consists of a script, template and optionally CSS styles. The script defines the state of the component and its methods. The template is an HTML structure enhanced with conditions, loops and variables that binds to the state in the script. It can also contain other components (or itself, thanks to recursion).

The information flows from the parent components to their children via the so-called props. The binding is again handled by the framework. The children can propagate changes back to their parents through events.

[10] `https://reactjs.org/`

7. Experiments

An important aspect of any technology is if it has any real use case. In this chapter, we present two examples of using MM-evocat. Although it is not meant to be optimized (yet), we have conducted several experiments to ensure the used algorithms have the assumed time complexity. So we do not try to measure precisely how long each operation takes, instead we focus on how the time scales with the size of the data.

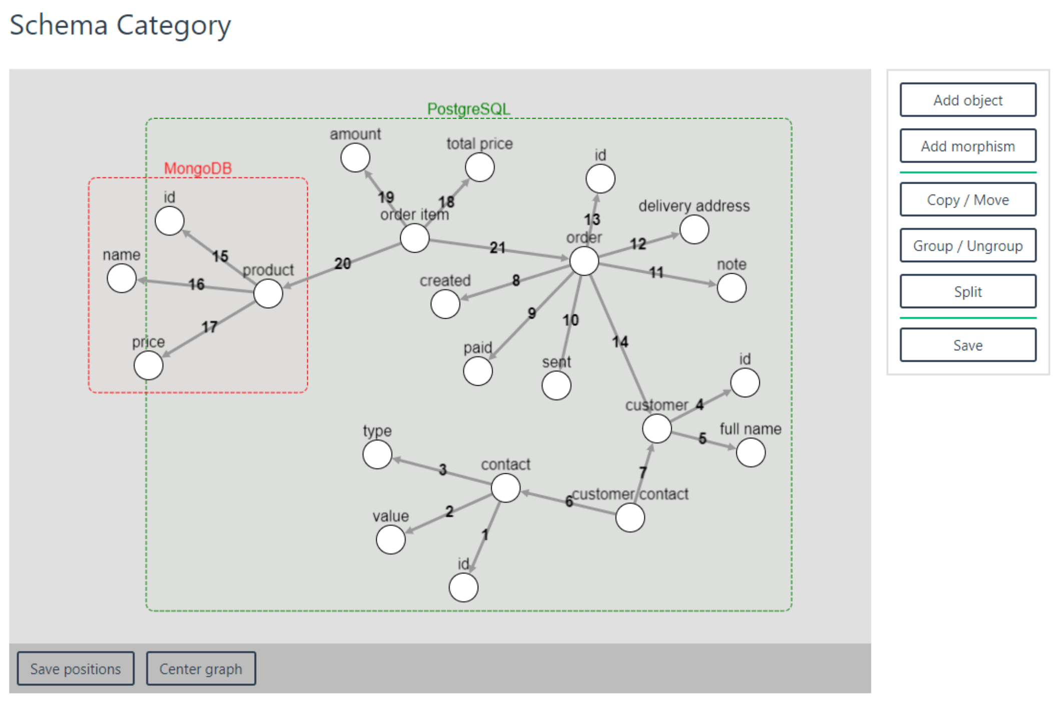


Figure 7.1: Screenshot of MM-cat with the schema category from the data migration experiment.

7.1 Cross-Model Data Migration

The schema category of the first example is depicted in Figure 7.1. There are several kinds (`customer`, `contact`, `customer contact`, `order`, `order item`), which are mapped to the corresponding tables in a relational database. There is also kind `product` stored in a document database. We want to migrate a subset of the data to a document database via the following mapping:

```
order: {
    address: 12,
    note: 11,
    customer: 14 {
        name: 5,
        contact: 6.-7 {
            <3>: 2
        }
    },
    events: {
        created: 8,
        sent: 9,
        paid: 10
    },
    items: -21 {
        amount: 19,
        total_price: 18,
        name: 16.20
    }
}
```

This example illustrates the key problem – for the given purpose we need data spread across multiple data sources. The main advantage of the framework is the ability to unify the view of the data from any combination of logical models to one schema, from which it can then be exported to any model.

7.1.1 Experiment

In the database, we had randomly generated data[1] for each respective input kind. There were 10^3 customers and each had on average 2 contacts. There were also 10^3 distinct products. The number of orders was the variable, describing the size of the output data, each of which had on average 3 items.

We measured the total time of each import (all kinds were imported separately) and export. In Figure 7.2, we can see the time of importing `orders` depending on the total amount of them. As we can see, the dependency is almost linear, which is what we would expect. We fitted it with a quadratic function

$$f\colon t = a_1 n + a_2 n^2\,, \tag{7.1}$$

the coefficients of the fit are

$$a_1 = (14.7 \pm 0.3) \cdot 10^{-5}\ \mathrm{s}\,, \tag{7.2}$$
$$a_2 = (1.8 \pm 1.0) \cdot 10^{-10}\ \mathrm{s}\,, \tag{7.3}$$

Where the values after $\pm$ represent the standard deviations. In the same Figure, we depicted the measured times of the exports. It is, again, almost linear dependency, so we fitted it with

$$g\colon t = b_1 n + b_2 n^2\,, \tag{7.4}$$

with the coefficients

$$b_1 = (7.2 \pm 0.3) \cdot 10^{-5}\ \mathrm{s}\,, \tag{7.5}$$
$$b_2 = (0.54 \pm 0.15) \cdot 10^{-10}\ \mathrm{s}\,. \tag{7.6}$$

The experiment concludes that over the span of several orders of magnitude, the time of imports and exports scales linearly with the size of the data. At the end of the scale, we can see a slight deviation from the linear trend, which is probably caused by the increasing memory efficiencies.

[1]`https://app.json-generator.com/`

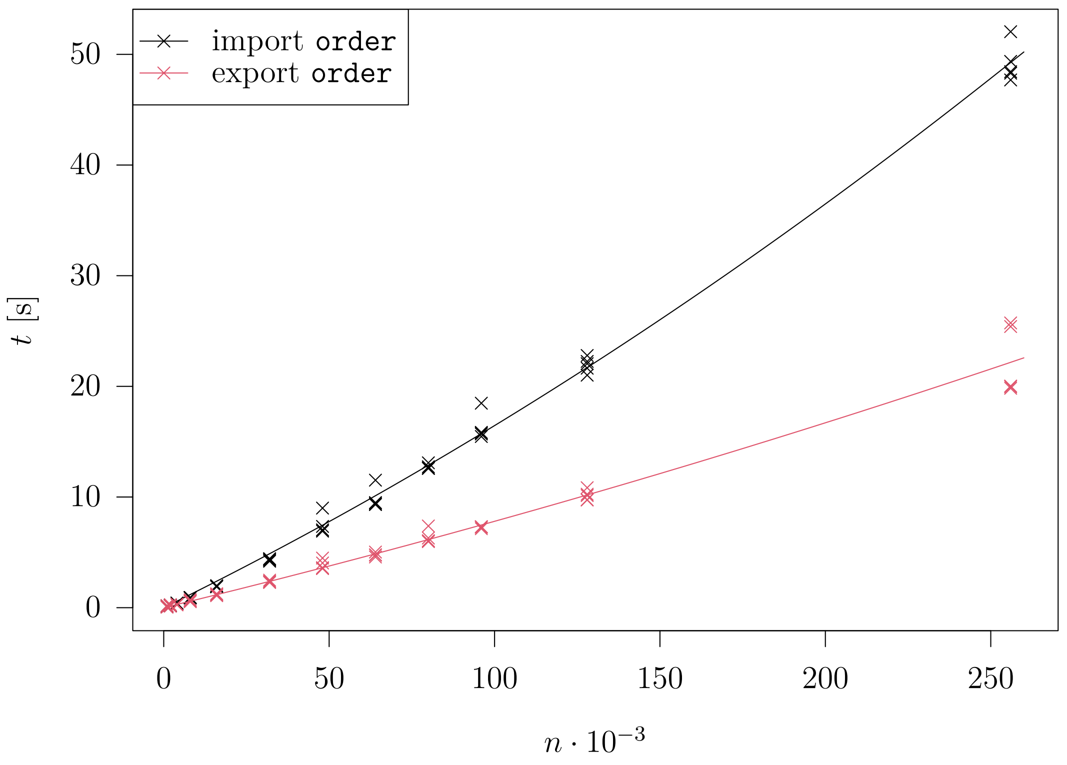


Figure 7.2: Time t to execute n cross-model data migration operations of kind `order` from the schema in Figure 7.1. The dependencies are fitted by the quadratic functions (see Equations (7.1) and (7.4)).

7.2 Complex Evolution Operations

The schema of the second example is depicted in Figure 7.3. The goal was to join the `street` with the `city` to create a `full address`, which would be then move to the `order` object. Because there are more orders than users, the addresses will be replicated. The point of this simple example is to test the complex operations. They are not yet implemented in a general way. However, we need to make sure they are feasible first.

7.2.1 Experiment

The data was generated in the same way as in the experiment from Section 7.1. The number of users was variable while each had on average 4.5 orders. We measured separately the time of the `JOIN/SPLIT` and `MOVE` operations. The results are depicted in Figure 7.4. Again, both dependencies look linear, so we fitted them with functions

$$f\colon t = a_1 n + a_2 n^2 \,, \tag{7.7}$$

$$g\colon t = b_1 n + b_2 n^2 \,. \tag{7.8}$$

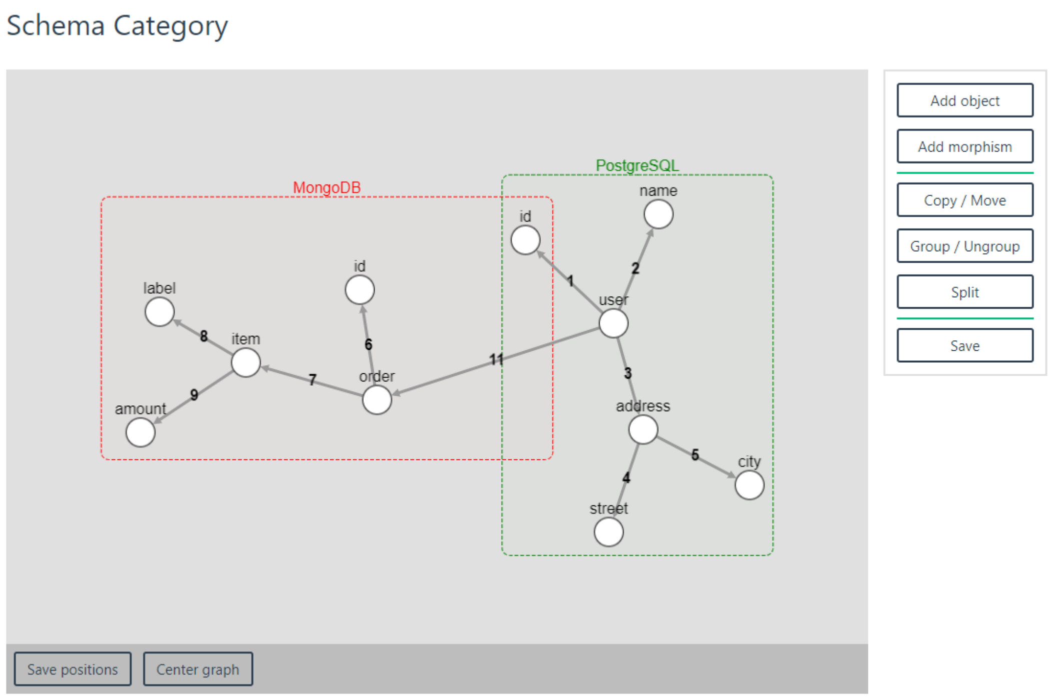


Figure 7.3: Screenshot of MM-cat with the schema category from the experiment with complex operations.

The fit coefficients are

$$a_1 = (1.8 \pm 0.1) \cdot 10^{-5}\ \mathrm{s}\,, \tag{7.9}$$
$$a_2 = (-0.2 \pm 0.1) \cdot 10^{-10}\ \mathrm{s}\,, \tag{7.10}$$
$$b_1 = (9.7 \pm 0.4) \cdot 10^{-5}\ \mathrm{s}\,, \tag{7.11}$$
$$b_2 = (-1.6 \pm 0.4) \cdot 10^{-10}\ \mathrm{s}\,. \tag{7.12}$$

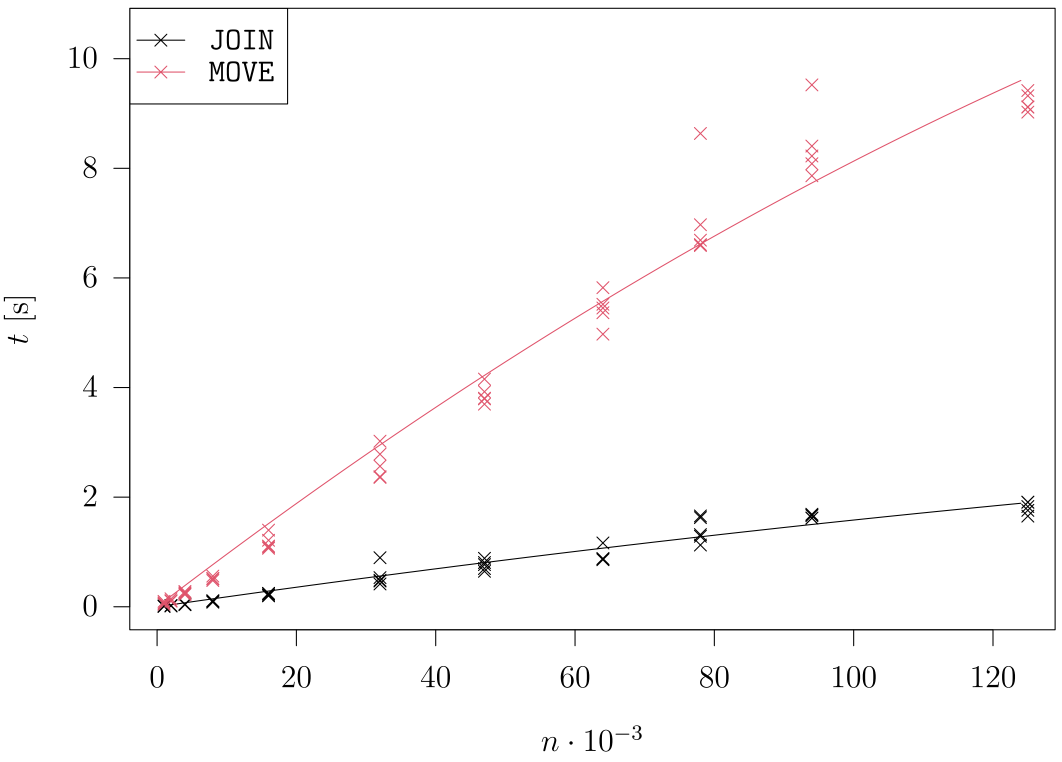


Figure 7.4: Time t to execute n complex `JOIN/SPLIT` and `MOVE` operations of `full address` in the schema depicted in Figure 7.3. The dependencies are fitted by the quadratic functions (see Equations (7.7) and (7.8)).

Conclusion

This thesis proposes a new approach to the evolution management of multi-model databases. Let us recapitulate its essential concepts.

Firstly, we reviewed the previous takes on the topic. We decided to base our proposal on the categorical approach because of its generality and the ability to cover all existing data models. Unlike most of the other current approaches, we chose to unify the data models instead of dealing with them separately. This allows us to model the data in a truly multi-model way.

After a brief introduction to category theory and a definition of basic data structures, we proceeded by designing the workflow and SMOs of our approach. We decided on a two-level architecture with a platform-independent conceptual model (described by a schema category) in the top layer and multiple platform-specific logical models (defined by their mappings to the schema category) in the bottom one.

The user primarily edits the conceptual schema, while the system propagates these changes to the logical models. In order to do so, we designed a set of SMOs, which we divided into three groups by their complexity and abstractness. We also considered SMOs in the logical layer. These affect only the logical models and the mappings.

As a proof of concept, we implemented the MM-cat framework, which allows us to create the schema category and the mappings, manage various database systems, and run cross-model data migration jobs. It currently supports the PostgreSQL and MongoDB database systems. Lastly, we performed two experiments to prove both the data migration and the complex evolution operations are reliable and linearly scalable.

7.3 Future Work

Both evolution management and data modeling are quite broad research areas offering bountiful new and exciting topics. We plan to continue working on the following aspects.

The first step is to expand the framework to fully functional prototype of MM-evocat by implementing all the proposed SMOs. In addition, we will add support for a few other database systems. There are also endless opportunities for their optimization in the form of both improving the efficiency of current algorithms and developing entirely new ones.

Our approach currently does not consider data types, so everything is a string. However, because of its generality, it can be easily extended to support all common data types. Another important feature of modern database systems are the integrity constraints, which are also subject to evolution. We do consider them and we also already have available algorithms [9] that would allow us to handle them. Their implementation is one of our priorities in the future.

Evolution management partially overlaps with schema inference, querying over the schema and propagation of changes to the queries. New approaches to these topics, based on category theory, are already being developed. And we can incorporate them into the framework.

Bibliography

[1] Kristopher Brown, David Spivak, and Ryan Wisnesky. Categorical data integration for computational science. *Computational Materials Science*, 164: 127–132, 06 2019.

[2] 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.

[3] Alberto Hernández Chillón, Meike Klettke, Diego Sevilla Ruiz, and Jesús García Molina. A taxonomy of schema changes for nosql databases, 2022.

[4] Daniel Crha. *Unified Querying of Multi-Model Data.* Master thesis, Charles University, Czech Republic, 2022.

[5] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. *ACM SIGACT News*, 33, 11 2002.

[6] Irena Holubová, Michal Vavrek, and Stefanie Scherzinger. Evolution management in multi-model databases. *Data & Knowledge Engineering*, 136: 101932, 2021.

[7] ISO/IEC 9075-1:2016. Information technology — database languages — sql — part 1: Framework (sql/framework). Standard, International Organization for Standardization, Geneva, CH, 12 2016.

[8] Pavel Koupil. *Modeling and Management of Multi-Model Data.* Doctoral thesis, Charles University, Czech Republic, 2022.

[9] Pavel Koupil and Irena Holubová. A unified representation and transformation of multi-model data using category theory. *Journal of Big Data*, 9(1): 61, 2022.

[10] Pavel Koupil, Jáchym Bártík, and Irena Holubová. MM-evocat: A Tool for Modelling and Evolution Management of Multi-Model Data. (under review).

[11] Pavel Koupil, Sebastián Hricko, and Irena Holubová. MM-infer: A Tool for Inference of Multi-Model Schemas. In *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.

[12] Jiaheng Lu and Irena Holubová. Multi-model databases: A new journey to handle the variety of data. *ACM Comput. Surv.*, 52, 2019.

[13] Clifford Lynch. How do your data grow? *Nature*, (455):28–29, 2008.

[14] Joshua Meyers, David I. Spivak, and Ryan Wisnesky. Fast left kan extensions using the chase. *Journal of Automated Reasoning*, pages 1–40, 07 2022.

[15] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Managing schema evolution in nosql data stores, 2013.

[16] 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.

[17] Uta Störl and Meike Klettke. Darwin: A data platform for nosql schema evolution management and data migration. 2022.

[18] Martin Svoboda, Pavel Čontoš, and Irena Holubová. Categorical modeling of multi-model data: One model to rule them all. In Christian Attiogbé and Sadok Ben Yahia, editors, *Model and Data Engineering*, pages 190–198. Springer International Publishing, 2021.

[19] Chris Tuijn and Marc Gyssens. Cgood, a categorical graph-oriented object data model. *Theoretical Computer Science*, 160(1):217–239, 1996.

List of Figures

List of Tables

List of Abbreviations

ACID Atomicity, Consistency, Isolation, Durability. 6, 9

API Application Programming Interface. 8, 48, 49

BASE Basically Available, Soft state, Eventual consistency. 6, 9

CIM Computation-Independent Model. 15

CSS Cascading Style Sheets. 50

DBMS Database Management System. 1, 5, 6

ER Entity-Relationship. 17, 18, 59

HTML HyperText Markup Language. 50

JSON JavaScript Object Notation. 8, 19, 21, 47, 48

MDA Model-driven architecture. 15

MVC Model View Controller. 48

MVVM Model View View-Model. 49

NoSQL Not only SQL. 4, 5, 6, 10, 11, 12

PIM Platform-Independent Model. 15

PSM Platform-Specific Model. 15

RDF Resource Description Framework. 6, 13

REST Representational State Transfer. 48

SMO Schema Modification Operation. 10, 11, 12, 13, 24, 25, 26, 27, 49, 56, 59, 61

SQL Structured Query Language. 9, 62

XML eXtensible Markup Language. 8, 47

A. Transitions Between Cardinalites

In this Appendix, we derive the transitions of the cardinalities as depicted in Figure A.1 which is used in Section 5.1.8. As we mentioned there, we have 16 different cardinalities and 32 unique transitions between them. Whenever we relax the limits of the cardinalities (i.e., from `1` to `0` for a minimum and from `1` to `*` for a maximum), the transition can be done automatically, because the new cardinality always meets the conditions of the original one. This means we have only 16 non-trivial transitions left.

Let us first show the general thought process on example $(\mathtt{1..1} \leftrightarrow \mathtt{1..*})$ to $(\mathtt{1..1} \leftrightarrow \mathtt{1..1})$. As a reminder, we are referring to the cardinalities $(\overline{c} \leftrightarrow c)$ of morphisms $\overline{f}\colon B \to A$ and $f\colon A \to B$. We denote the elements from the instance objects as $a \in I(A)$ and $b \in I(B)$. For each of them, we define one of these auxiliary sets

$$f(a) = \{b \mid b \in I(B), (a, b) \in I(f)\}\,, \tag{A.1}$$

$$f(b) = \{a \mid a \in I(A), (a, b) \in I(f)\}\,. \tag{A.2}$$

Now back to the example. Some of the a elements have multiple b elements, while each b has exactly one a. We have two options:

- $\mathbf{B}^-$ – for each $a \in I(A)$, remove all but one $b \in f(a)$,
- $\mathbf{A}^+$ – for each $a \in I(A)$, for each $b \in f(a)$ except the first one, create a copy of a and link it to b.

The $\mathbf{B}^-$ solution allows the user to choose which b elements to keep and which to remove. We can represent it by function $\beta\colon f(a) \to b \in f(a)$ that for each a gets on the input all the $f(a)$ elements and then returns the one that should stay. On the other hand, the $\mathbf{A}^+$ option is straightforward – we just copy some elements and link them.

Now we will focus on the actual transitions in general. There are eight of them with the pattern $(\mathtt{0..}\nu \leftrightarrow \zeta\mathtt{..}\eta)$ to $(\mathtt{1..}\nu \leftrightarrow \zeta\mathtt{..}\eta)$. That represents the cases with some b elements without any corresponding a elements. We want to change it so that each b will have its a. Because the excess b elements are not linked to any a, we do not know anything about them, so we have no other option than to delete them.

That leaves us with just eight remaining transitions that share the pattern $(\mu\mathtt{..}\nu \leftrightarrow \zeta\mathtt{..*})$ to $(\mu\mathtt{..}\nu \leftrightarrow \zeta\mathtt{..1})$. An example of this is at the beginning of this Appendix. We have to determine when $\mathbf{B}^-$ and $\mathbf{A}^+$ solutions are applicable.

First, let us show that $\mathbf{A}^+$ has no limitations. We want to prove that, supposing all the original a and b elements satisfied the $(\mu\mathtt{..}\nu \leftrightarrow \zeta\mathtt{..*})$ conditions, the new a' and b' elements, which are the result of the transformation, meet the $(\mu\mathtt{..}\nu \leftrightarrow \zeta\mathtt{..1})$ requirements. All the new b' elements are the same as the old b ones, but some changed their links from the original a elements to their

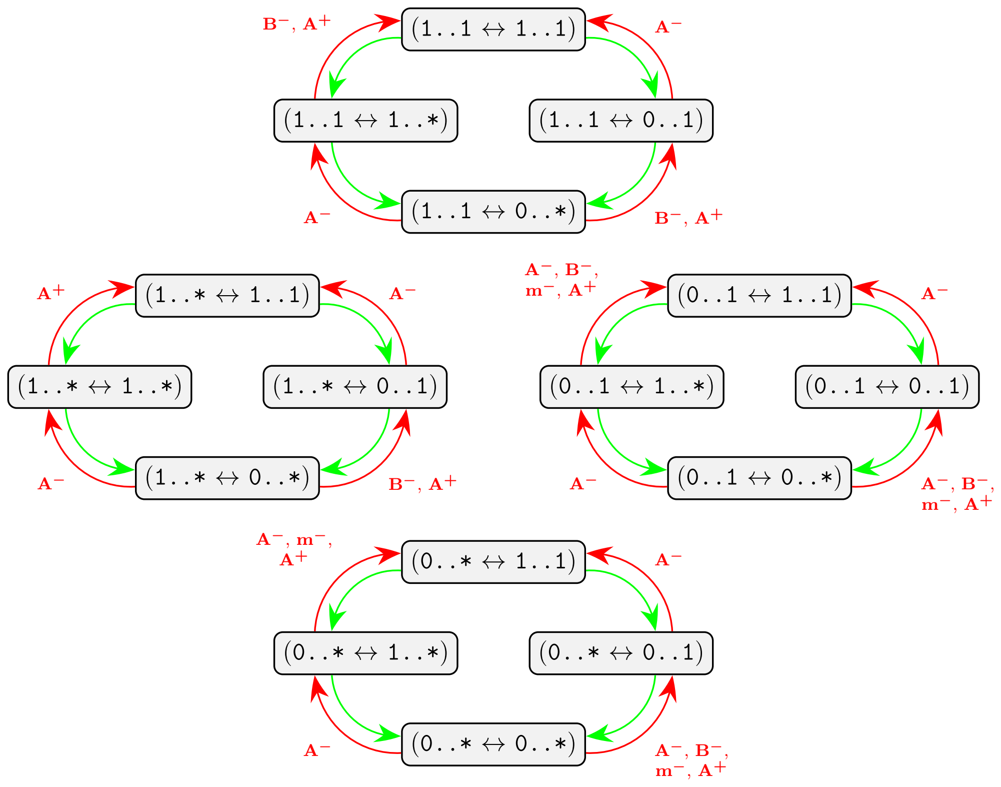


Figure A.1: All possible cardinalities and both trivial (green) and non-trivial (red) transitions between them. For better clarity, half of the possible transitions are missing. However, they all are just mirror images of those depicted here. See Figure A.2

copies. Those that did not change automatically satisfy the original $\mu\,..\,\nu$ condition. Those that did change now have exactly one a' linked to them, so they meet the `1..1` requirement. Because that is the most strict one, they certainly meet any condition, i.e., also $\mu\,..\,\nu$.

Some of the a' elements remained unchanged, except now they have only one b linked, so they satisfy $\zeta\,..$`1`. The other a', i.e., those that were added, have exactly one linked b'. Now we just use the same argument as for the changed b' elements and the proof is completed.

The method for the $\mathbf{B}^-$ solution is similar except for the ($\mu\,..$`*` $\leftrightarrow$ `1..*`) to ($\mu\,..$`*` $\leftrightarrow$ `1..1`) case. In other words, $\mathbf{B}^-$ is not applicable when each b can have multiple $a \in f(b)$ elements and each a has at least one $b \in f(a)$. The problem is that when we try to delete $b \in f(a_1)$ for some a_1, if there are multiple elements in $f(b)$, one of them (let us denote it as $a_2 \in f(b)$, where $a_2 \neq a_1$) might have only one b linked to it. Which is of course the b we are currently removing. At the end, the $f(a_2)$ set will be empty, which contradicts the minimum of the `1..1` condition.

In general, there are two more solutions than only $\mathbf{B}^-$ and $\mathbf{A}^+$. We can do:

- $\mathbf{A}^-$ – for each a, remove it if $f(a)$ set has more than one element,
- $\mathbf{m}^-$ – for each a, for each $b \in f(a)$ except the first one, remove the (a, b) link from $I(f)$ (but keep both a and b).

The $\mathbf{A}^-$ solution is automatic, while the $\mathbf{m}^-$ one is controlled by the same function β as the $\mathbf{B}^-$ one. Both $\mathbf{A}^-$ and $\mathbf{m}^-$ are applicable in all cases except those with the $(\mathtt{1..}\nu \leftrightarrow \zeta\mathtt{..*})$ to $(\mathtt{1..}\nu \leftrightarrow \zeta\mathtt{..1})$ pattern, i.e., when each b' has at least one $a' \in f(b')$. The reasons are as follows:

- $\mathbf{A}^-$ – deleting each a that has more than one $b_1 \in f(a)$ can cause a specific b_2 to lose all its linked a elements,
- $\mathbf{m}^-$ – deleting the links can lead to the same problem.

Finally, all 16 non-trivial transitions are solved, giving us 32 unique transitions in total. The other 32, i.e., their mirror images, can be obtained by substituting A with B (and $\mathbf{A}^+$ with $\mathbf{B^1}$ and so on). Some of them are shown in Figure A.2.

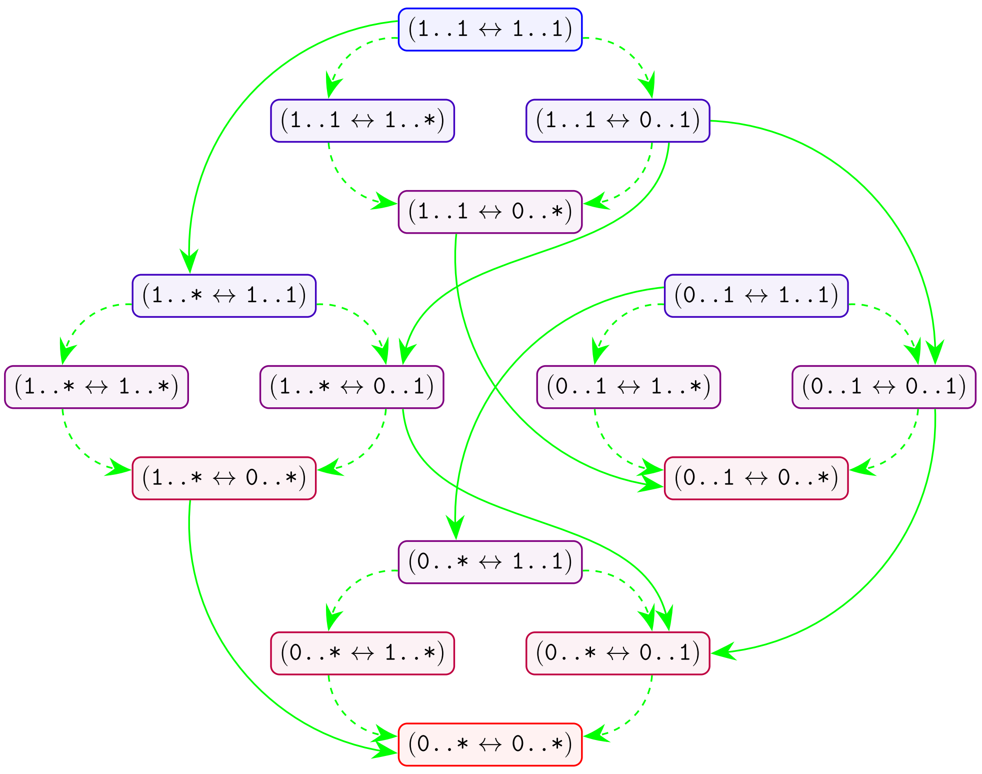


Figure A.2: The original transitions from Figure A.1 (dashed) plus eight of the additional ones (solid). Note that, for better clarity, only the trivial transitions are displayed. The non-trivial ones would be exactly the same except in the opposite direction. Also, the colors of the nodes correspond to how many trivial transitions originate in them, from blue (four transitions) to red (zero).

A.1 Coexistence of Morphisms

Last but not least, we must find out which morphisms can coexist together. Let us consider two morphisms between the same objects, i.e., $f_1\colon A \to B$ and $f_2\colon A \to B$. If the cardinality of the first one was $(\mathtt{1..1} \leftrightarrow \mathtt{1..1})$, then the second one can not be, for example, $(\mathtt{1..1} \leftrightarrow \mathtt{1..*})$, because there are not enough $b \in I(B)$ so that at least $a \in I(A)$ could have more than one of them.

As we can see, the reason is that some cardinalities compare the number of elements in $I(A)$ and $I(B)$. Let us denote these numbers as $|I(A)|$ and $|I(B)|$. Then we can divide the cardinalities into the following categories:

- $|I(A)| = |I(B)|$ – (`1..1` $\leftrightarrow$ `1..1`),
- $|I(A)| \leq |I(B)|$ – (`0..1` $\leftrightarrow$ `1..1`), (`1..1` $\leftrightarrow$ `1..*`), (`0..1` $\leftrightarrow$ `1..*`),
- $|I(A)| \geq |I(B)|$ – (`1..1` $\leftrightarrow$ `0..1`), (`1..*` $\leftrightarrow$ `1..1`), (`1..*` $\leftrightarrow$ `0..1`),
- it is impossible to tell – all others.

Let us first focus only on the first three groups. If we put a morphism from one of them next to a morphism from another, the resulting relation between $|I(A)|$ and $|I(B)|$ must be an equality[1] But that means both morphisms should have the (`1..1` $\leftrightarrow$ `1..1`) cardinality instead. So we can conclude that any two morphisms between the first three groups are not compatible. Morphisms within the same group are always compatible.

Finally, we will look at the last group. The cardinalities of its morphisms do not state anything about the relation of $|I(A)|$ and $|I(B)|$, so they are compatible with all other morphisms.

[1]For example, $|I(A)| \leq |I(B)| \wedge |I(A)| \geq |I(B)| \Rightarrow |I(A)| = |I(B)|$.