MM-evoque: Query Synchronisation in Multi-Model Databases*

Pavel Koupil
Department of Software Engineering, Charles University
Prague, Czech Republic
pavel.koupil@matfyz.cuni.cz

Jáchym Bártík
Department of Software Engineering, Charles University
Prague, Czech Republic
jachym.bartik@matfyz.cuni.cz

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

ABSTRACT

As multi-model databases become increasingly prevalent, the evolving multi-model schemas pose a significant challenge to query validity. This demo paper introduces *MM-evoque*, a novel tool designed to propagate schema changes across multi-model databases and synchronise not only data but also respective queries to remain valid and performant.

To cover all the popular models and their combinations, *MM-evoque* is based on the categorical representation of multi-model data. To cover non-standard multi-model query languages of the underlying database systems, we use a conceptual multi-model query language MMQL. This demo paper introduces the functionality of *MM-evoque* in several types of use cases.

1 INTRODUCTION

Unlocking the power of multi-model database management systems (DBMSs)[1] is not just about keeping pace with the present majority; it is a strategic leap into the future of data complexity. Regrettably, our extensive survey [11] reveals significant variations in multi-model database management systems (DBMSs) even in the core aspects – the supported models, the way they are combined, as well as the multi-model query languages. This situation brings many new challenges to multi-model data management. We have already proposed several solutions based on the categorical representation of multi-model data, the so-called *schema category* [8]. This abstract graph representation backed by the formalism of category theory enabled us to build a family of tools for modelling and transformations [10], schema inference [9], data migration under evolving schema [5], or querying using SPARQL-like query language MMQL [6].

In this paper, we further extend our toolset with a new complementary member called *MM-evoque*[2]. It extends the functionalities we already provide with an important novel feature – propagation of changes in the multi-model schema to respective queries. The main contributions can be summarised as follows:

- Since we use the unifying categorical representation, we support any combination of existing popular data models (relational, key/value, document, column, array, and graph) that can be mutually embedded, referenced, or overlapped (i.e., redundancy is naturally supported too).
- The graph categorical representation can be queried using a SPARQL-like query language MMQL that is then decomposed and translated to a set of Database-Specific Languages (DSLs) of underlying DBMSs where they are executed. The results are combined into the final result.
- We introduce a set of schema-modification operations (SMOs) to modify the categorical schema. As each schema is represented as a sequence of operations creating it, the user can return to any previous version.
- The schema changes are propagated to underlying data and respective queries over the data. We ensure synchronisation of the queries when needed and possible. If the correction is ambiguous or cannot be done automatically, the user is informed at least.
- We present a prototype implementation of *MM-evoque* demonstrating the indicated advantages in several use cases and shielding the user from the implementation specifics of particular DBMSs.

Paper Outline. In Section 2, we review related work. Section 3 introduces the categorical representation of multi-model data and the categorical query language MMQL. Section 4 describes the presented tool and a bigger example. In Section 5, we outline its demonstration.

2 RELATED WORK

The evolution management is challenging even for a single model. There exist approaches, e.g., for relational DBMSs [13], NoSQL systems [14], and first approaches for multi-model DBMSs [5]. But they focus primarily on data adaptation/migration.

Considering also queries, we can encounter related approaches with different aims and motivations. *Query rewriting* [4] focuses on transforming a query to a more efficient version. *Query adaptation* [2] focuses on adapting query plans to changing environmental conditions at runtime. In contrast, in this paper, we aim at *query synchronisation* reflecting the evolution of the schema, in particular of multi-model data. Several approaches also exist here, though subtly less represented. Most of them focus on a single model (relational or XML) [1]; some consider aggregate-oriented NoSQL models [12], or polystores [3].

From its beginning, our approach targets all popular models and existing types of their combinations. It provides an intuitive conceptual graph layer and SPARQL-like querying, both covertly backed by the category theory.

3 CATEGORICAL MODEL AND QUERYING

Let us first remember the basic notions of category theory. A *category* $\mathbf{C} = (\mathcal{O}, \mathcal{M}, \circ)$ consists of a set of objects $\mathcal{O}$, set of morphisms $\mathcal{M}$, and a composition operation $\circ$ over the morphisms ensuring transitivity and associativity. Each morphism is modelled as an arrow $f : A \rightarrow B$, where $A, B \in \mathcal{O}$, $A = dom(f)$, $B = cod(f)$. And there is an *identity* morphism $1_A \in \mathcal{M}$ for each object A. The key aspect is that a category can be visualised as a multigraph, where objects act as vertices and morphisms as directed edges.

Schema Category. The *schema category* [8] forms the core of conceptual modelling in all our tools. It is defined as a tuple $\mathbf{S} = (\mathcal{O}_\mathbf{S}, \mathcal{M}_\mathbf{S}, \circ_\mathbf{S})$. Objects in $\mathcal{O}_\mathbf{S}$ correspond to the domains of

*Supported by the GAČR project no. 23-07781S and GAUK project no. 292323.
[1] https://db-engines.com/en/ranking
[2] https://www.ksi.mff.cuni.cz/~koupil/mm-evoque/ (including a demo video)

the ER model's entity types, attributes, and relationship types. Morphisms (of semantics "has a property", "has an identifier", "has a role", or "is a") in $\mathcal{M}_S$ connect appropriate pairs of objects. The explicitly defined morphisms are denoted as *base*, obtained via the composition $\circ$ as *composite*.

Since the terminology within the particular models differs, we use a unification of respective model-specific terms [8]. A *kind* corresponds to a class of items represented in each model, e.g., a relational table or a collection of JSON documents. A *record* corresponds to one item of a kind, e.g., a table row or a JSON document. Depending on a particular model, a record consists of simple or complex *properties* having their *domains*.

The decomposition of a schema category S, eventually partial or overlapping, is defined via a set of *mappings* [8]. For each kind, the mapping specifies where and how its records are stored in a selected single-/multi-model DBMS using a so-called *access path*, which recursively describes the structure of a kind.

Example 3.1. In Figure 5 on the left, we can see a schema category describing customers, their friendships, products, and orders. It is decomposed and mapped to relational (violet) and document (green) models. □

Multi-Model Query Language (MMQL). The MMQL [6] is a SPARQL-like model-agnostic query language based on pattern matching. The MMQL expression consists of a projection (`SELECT`) that defines the structure of the result and a selection (`WHERE`) where graph patterns can be defined along with filter conditions (`FILTER`). We also support common constructs such as `ORDER BY`, `LIMIT`, or aggregation functions. Graph patterns are expressed using triples $p : s \longrightarrow o$, where o, s (e.g., variables or constants) correspond to objects from $\mathcal{O}_S$ and p maps to morphisms in $\mathcal{M}_S$. For query q, patterns in selection form a *selection category* S_q^σ (a subcategory of S) and patterns in projection form a *projection category* S_q^π (a subcategory of S_q^σ).

Example 3.2. Figure 5 on the right shows two MMQL queries q_1: "*For each pair of friends, return their names and surnames*" and q_2: "*Find the most expensive order ordered by Anne and return its price*". □

Multi-Model Schema Evolution Language (MMSEL). The MMSEL [5] consists of 8 basic *schema modification operations* (SMOs) – see Table 1 – that enable the gradual creation (and deletion) of any *consistent* schema category. Hence, each operation is assigned with prerequisites that must be fulfilled.

Table 1: Basic SMOs

SMO	Behaviour	Prerequisitity
$create(O)$	Creates object O with a unique identifier and default parameters.	–
$create(m, D, C)$	Creates morphism $m : D \longrightarrow C$ with a unique signature and default parameters.	$D, C \in \mathcal{O}_S$
$delete(O)$	Deletes object O.	$O \in \mathcal{O}_S$, O can participate only in $1_O \in \mathcal{M}_S$
$delete(m)$	Deletes morphism m.	$m \in \mathcal{M}_S$
$rename(O, n)$	Changes the name of object O to n.	$O \in \mathcal{O}_S$
$move(m, p)$	Sets $dom(m)$ to $cod(p[last])$, i.e. to the end of a path p.	$m \in \mathcal{M}_S$, $dom(p[first]) = dom(m)$
$copy(O)$	Creates a copy O' of object O (with a new unique identifier)	$O \in \mathcal{O}_S$
$copy(m, D, C)$	Creates a copy $m' : D \longrightarrow C$ of morphism m (with a new unique signature)	$D, C \in \mathcal{O}_S$, $D = copy(dom(m))$, $C = copy(cod(m))$

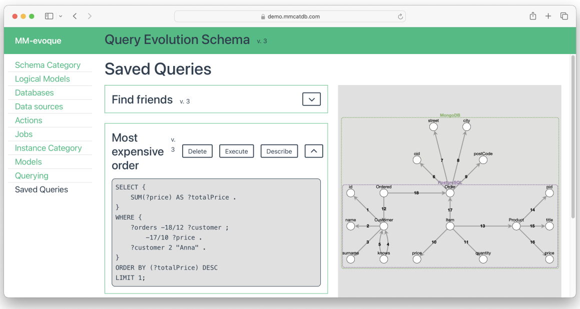
In addition, we have operations *addMapping*() and *deleteMapping*(), which enable one to add/delete mapping of a kind, i.e., data migration.

Basic SMOs form more user-friendly *composite SMOs*, such as, e.g., (un)group or join/split [5]. *Correct* composition ensures that we only need to define the propagation of basic SMOs.

4 *MM-EVOQUE*

MM-evoque was built on categorical representation, MMQL, and MMSEL to support query synchronisation. Currently, it supports the following models/DBMSs: *PostgreSQL*[3] (relational and document, i.e., multi-model), *Neo4j*[4] (graph), *MongoDB*[5] (JSON document), and *Apache Cassandra*[6] (columnar model).

In Figure 1, we provide a screenshot of *MM-evoque*. On the left, we can see MMQL queries synchronised with the schema category from Figure 5 visualised on the right.



```
SELECT {
    SUM(?price) AS ?totalPrice .
}
WHERE {
    ?orders -18/12 ?customer ;
        -17/10 ?price .
    ?customer 2 "Anna" .
}
ORDER BY (?totalPrice) DESC
LIMIT 1;
```

Figure 1: A screenshot of *MM-evoque*

Example 4.1. Consider Figures 2–5 and queries q_1, q_2 from Example 3.2.

Figure 2 shows a naive relational schema with several redundant values in table *Orders* (version 1). We also depict the mapping of S to the relational model, where the violet objects represent kinds (tables), and their neighbours represent their properties (columns). The efficiency of q_1 and q_2 is satisfactory for low numbers of customers and products.

In Figure 3, we decided on the normalisation of the data. We applied the respective SMOs to S (version 2) and triggered their propagation to data [5]. Their propagation to queries ensured by *MM-evoque* does not change q_1. But for q_2, it adds the new relationship tables to the selection of MMQL; thus, the SQL translation contains more joins.

In Figure 4 (version 3A), we try to use the document model instead. Hence, the schema category S remains unchanged, but the data is migrated from the relational to the document model (green). Queries q_1 and q_2 are slightly (technically) modified since we assume that every graph pattern must start in the root of a kind. However, we still have a problem with data redundancy (friends of customers and products) in this solution.

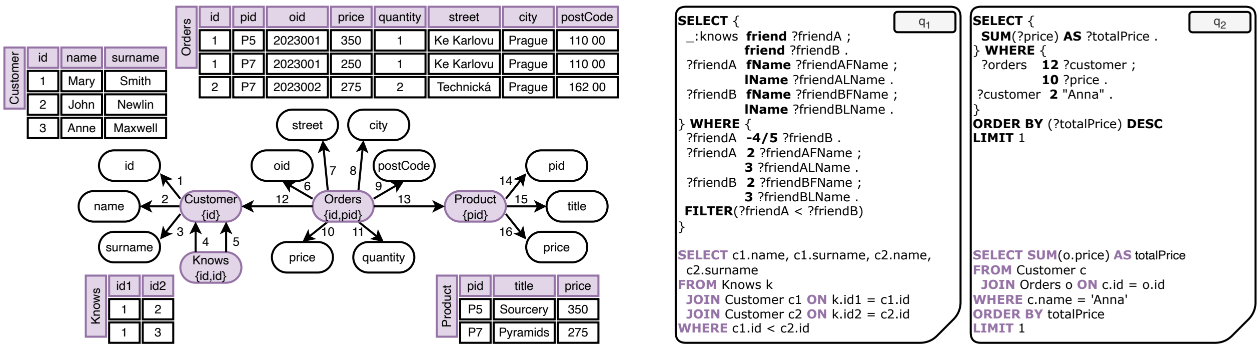
In Figure 5 (version 3B), we utilise the multi-model paradigm and migrate the data to a combination of relational (violet) and document (green) models. q_1 and q_2 are synchronised, respectively, and we gain a good performance for both. □

4.1 Workflow

Having the old and new versions of the schema category, denoted S and S′, and the MMQL query q to be synchronised with the changes, the workflow of *MM-evoque* is as follows:

(1) The sequence of basic SMOs which changed S to S′ is determined (as composite SMOs consist of basic ones).

[3] https://www.postgresql.org/
[4] https://neo4j.com/
[5] https://www.mongodb.com/
[6] https://cassandra.apache.org/_/index.html



Customer

id	name	surname
1	Mary	Smith
2	John	Newlin
3	Anne	Maxwell

Orders

id	pid	oid	price	quantity	street	city	postCode
1	P5	2023001	350	1	Ke Karlovu	Prague	110 00
1	P7	2023001	250	1	Ke Karlovu	Prague	110 00
2	P7	2023002	275	2	Technická	Prague	162 00

Knows

id1	id2
1	2
1	3

Product

pid	title	price
P5	Sourcery	350
P7	Pyramids	275

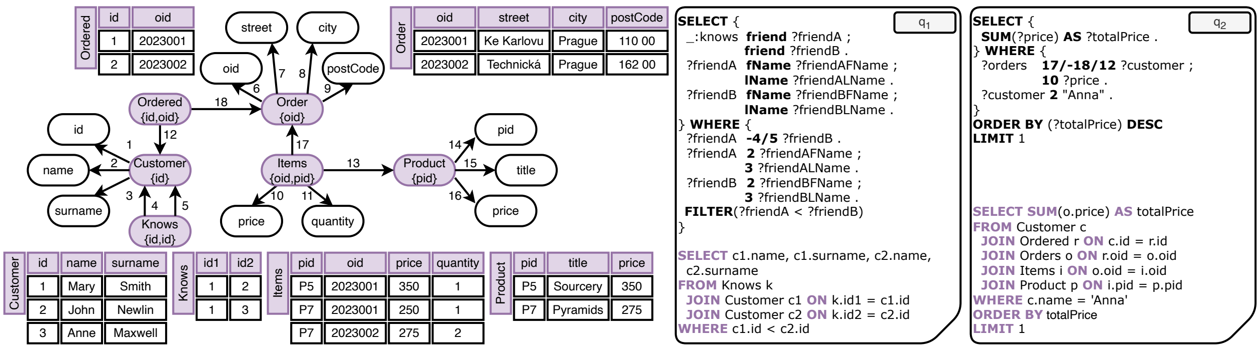
```
SELECT {
  _:knows  friend ?friendA ;
           friend ?friendB .
  ?friendA  fName ?friendAFName ;
            lName ?friendALName .
  ?friendB  fName ?friendBFName ;
            lName ?friendBLName .
} WHERE {
  ?friendA  -4/5 ?friendB .
  ?friendA  2 ?friendAFName ;
            3 ?friendALName .
  ?friendB  2 ?friendBFName ;
            3 ?friendBLName .
 FILTER(?friendA < ?friendB)
}

SELECT c1.name, c1.surname, c2.name,
  c2.surname
FROM Knows k
  JOIN Customer c1 ON k.id1 = c1.id
  JOIN Customer c2 ON k.id2 = c2.id
WHERE c1.id < c2.id
```

```
SELECT {
  SUM(?price) AS ?totalPrice .
} WHERE {
  ?orders   12 ?customer ;
            10 ?price .
 ?customer  2 "Anna" .
}
ORDER BY (?totalPrice) DESC
LIMIT 1

SELECT SUM(o.price) AS totalPrice
FROM Customer c
  JOIN Orders o ON c.id = o.id
WHERE c.name = 'Anna'
ORDER BY totalPrice
LIMIT 1
```

Figure 2: Version 1 – a naive mapping to the relational model (PostgreSQL)



Ordered

id	oid
1	2023001
2	2023002

Order

oid	street	city	postCode
2023001	Ke Karlovu	Prague	110 00
2023002	Technická	Prague	162 00

Customer

id	name	surname
1	Mary	Smith
2	John	Newlin
3	Anne	Maxwell

Knows

id1	id2
1	2
1	3

Items

pid	oid	price	quantity
P5	2023001	350	1
P7	2023001	250	1
P7	2023002	275	2

Product

pid	title	price
P5	Sourcery	350
P7	Pyramids	275

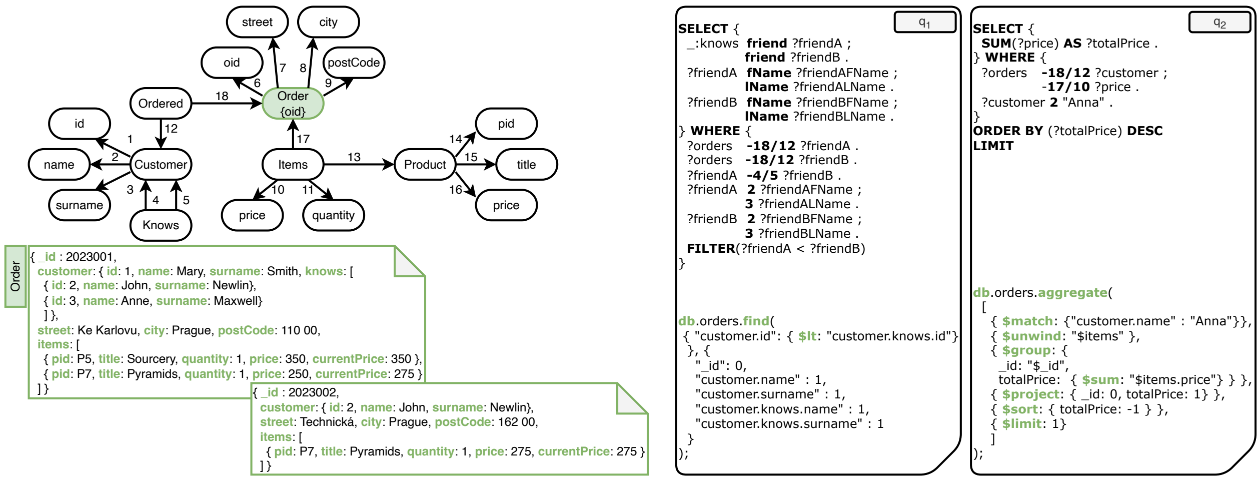
```
SELECT {
  _:knows  friend ?friendA ;
           friend ?friendB .
  ?friendA  fName ?friendAFName ;
            lName ?friendALName .
  ?friendB  fName ?friendBFName ;
            lName ?friendBLName .
} WHERE {
  ?friendA  -4/5 ?friendB .
  ?friendA  2 ?friendAFName ;
            3 ?friendALName .
  ?friendB  2 ?friendBFName ;
            3 ?friendBLName .
 FILTER(?friendA < ?friendB)
}

SELECT c1.name, c1.surname, c2.name,
  c2.surname
FROM Knows k
  JOIN Customer c1 ON k.id1 = c1.id
  JOIN Customer c2 ON k.id2 = c2.id
WHERE c1.id < c2.id
```

```
SELECT {
  SUM(?price) AS ?totalPrice .
} WHERE {
  ?orders   17/-18/12 ?customer ;
            10 ?price .
  ?customer 2 "Anna" .
}
ORDER BY (?totalPrice) DESC
LIMIT 1

SELECT SUM(o.price) AS totalPrice
FROM Customer c
  JOIN Ordered r ON c.id = r.id
  JOIN Orders o ON r.oid = o.oid
  JOIN Items i ON o.oid = i.oid
  JOIN Product p ON i.pid = p.pid
WHERE c.name = 'Anna'
ORDER BY totalPrice
LIMIT 1
```

Figure 3: Version 2 – a mapping to a normalized relational schema (PostgreSQL)



```
{ _id : 2023001,
  customer: { id: 1, name: Mary, surname: Smith, knows: [
   { id: 2, name: John, surname: Newlin},
   { id: 3, name: Anne, surname: Maxwell}
   ] },
  street: Ke Karlovu, city: Prague, postCode: 110 00,
  items: [
   { pid: P5, title: Sourcery, quantity: 1, price: 350, currentPrice: 350 },
   { pid: P7, title: Pyramids, quantity: 1, price: 250, currentPrice: 275 }
  ] }
```

```
{ _id : 2023002,
  customer: { id: 2, name: John, surname: Newlin},
  street: Technická, city: Prague, postCode: 162 00,
  items: [
   { pid: P7, title: Pyramids, quantity: 1, price: 275, currentPrice: 275 }
  ] }
```

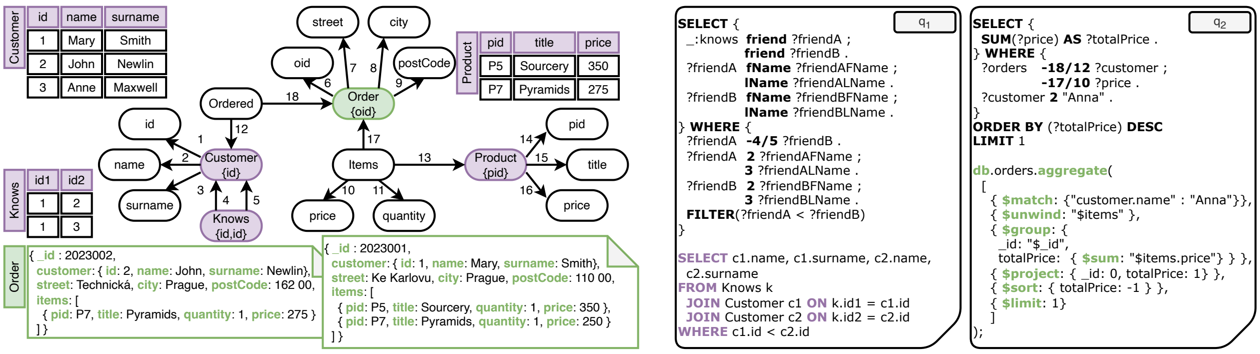
```
SELECT {
  _:knows  friend ?friendA ;
           friend ?friendB .
  ?friendA  fName ?friendAFName ;
            lName ?friendALName .
  ?friendB  fName ?friendBFName ;
            lName ?friendBLName .
} WHERE {
  ?orders   -18/12 ?friendA .
  ?orders   -18/12 ?friendB .
  ?friendA  -4/5 ?friendB .
  ?friendA  2 ?friendAFName ;
            3 ?friendALName .
  ?friendB  2 ?friendBFName ;
            3 ?friendBLName .
  FILTER(?friendA < ?friendB)
}

db.orders.find(
 { "customer.id": { $lt: "customer.knows.id"}
  }, {
   "_id": 0,
   "customer.name" : 1,
   "customer.surname" : 1,
   "customer.knows.name" : 1,
   "customer.knows.surname" : 1
  }
);
```

```
SELECT {
  SUM(?price) AS ?totalPrice .
} WHERE {
  ?orders   -18/12 ?customer ;
            -17/10 ?price .
  ?customer 2 "Anna" .
}
ORDER BY (?totalPrice) DESC
LIMIT

db.orders.aggregate(
  [
   { $match: {"customer.name" : "Anna"}},
   { $unwind: "$items" },
   { $group: {
     _id: "$_id",
     totalPrice:  { $sum: "$items.price"} } },
   { $project: { _id: 0, totalPrice: 1} },
   { $sort: { totalPrice: -1 } },
   { $limit: 1}
   ]
);
```

Figure 4: Version 3A – a mapping to the document model (MongoDB)



Customer

id	name	surname
1	Mary	Smith
2	John	Newlin
3	Anne	Maxwell

Product

pid	title	price
P5	Sourcery	350
P7	Pyramids	275

Knows

id1	id2
1	2
1	3

```
{ _id : 2023002,
  customer: { id: 2, name: John, surname: Newlin},
  street: Technická, city: Prague, postCode: 162 00,
  items: [
   { pid: P7, title: Pyramids, quantity: 1, price: 275 }
  ] }
```

```
{ _id : 2023001,
  customer: { id: 1, name: Mary, surname: Smith},
  street: Ke Karlovu, city: Prague, postCode: 110 00,
  items: [
   { pid: P5, title: Sourcery, quantity: 1, price: 350 },
   { pid: P7, title: Pyramids, quantity: 1, price: 250 }
  ] }
```

```
SELECT {
  _:knows  friend ?friendA ;
           friend ?friendB .
  ?friendA  fName ?friendAFName ;
            lName ?friendALName .
  ?friendB  fName ?friendBFName ;
            lName ?friendBLName .
} WHERE {
  ?friendA  -4/5 ?friendB .
  ?friendA  2 ?friendAFName ;
            3 ?friendALName .
  ?friendB  2 ?friendBFName ;
            3 ?friendBLName .
  FILTER(?friendA < ?friendB)
}

SELECT c1.name, c1.surname, c2.name,
  c2.surname
FROM Knows k
  JOIN Customer c1 ON k.id1 = c1.id
  JOIN Customer c2 ON k.id2 = c2.id
WHERE c1.id < c2.id
```

```
SELECT {
  SUM(?price) AS ?totalPrice .
} WHERE {
  ?orders   -18/12 ?customer ;
            -17/10 ?price .
  ?customer 2 "Anna" .
}
ORDER BY (?totalPrice) DESC
LIMIT 1

db.orders.aggregate(
  [
   { $match: {"customer.name" : "Anna"}},
   { $unwind: "$items" },
   { $group: {
     _id: "$_id",
     totalPrice:  { $sum: "$items.price"} } },
   { $project: { _id: 0, totalPrice: 1} },
   { $sort: { totalPrice: -1 } },
   { $limit: 1}
   ]
);
```

Figure 5: Version 3B – a multi-model mapping (PostgreSQL and MongoDB)

Table 2: Basic SMOs and their impact on MMQL query constructs

	$create(O)$	$create(m, D, C)$	$delete(O)$	$delete(m)$	$rename(O, n)$	$move(m, p)$	$copy(O)$	$copy(m, D, C)$	$addMapping()$ $deleteMapping()$
S_q^σ	0	0	$0 \mid \infty$	$0 \mid 1 \mid \infty$	0	0	0	0	$0 \mid 1$
`VALUES`	0	0	$0 \mid \infty$	$0 \mid 1$	0	0	0	0	$0 \mid 1$
`FILTER`	0	0	$0 \mid \infty$	$0 \mid 1$	0	0	0	0	$0 \mid 1$
S_q^π	0	0	$0 \mid \infty$	$0 \mid 1$	0	0	0	0	$0 \mid 1$
`DISTINCT`	0	0	$0 \mid \infty$	$0 \mid 1$	0	1	0	0	$0 \mid 1$
`ORDER BY`	0	0	$0 \mid \infty$	$0 \mid 1$	0	0	0	0	$0 \mid 1$
`LIMIT`	0	0	0	0	0	0	0	0	$0 \mid 1$
Aggregations	0	0	$0 \mid \infty$	$0 \mid \infty$	0	∞	0	0	$0 \mid 1$

(2) MMQL query q is translated into a set of constructs. The selection category S_q^σ and projection category S_q^π are created according to respective graph patterns of q.
(3) Using Table 2, we determine whether q can be synchronised as $max(impact(smo, constuct))$ from all pairs (SMO, affected MMQL construct) occurring in q. If the result is 0, we can synchronise q and preserve its original behaviour. If the result is 1, we can synchronise q, but its behaviour may change (see Section 4.2). Otherwise, the changes are too critical that we cannot easily adapt the query.
(4) If the user agrees with the proposed changes, we apply the SMOs to S_q^σ and S_q^π. We use the fact that both are subcategories of S, so we identify SMOs that affect them.
(5) The modified query q' is translated to a new version of DSL queries.

4.2 SMO-to-MMQL Propagation

Let us discuss the impact of SMOs from Table 1 to S_q^σ and S_q^π.

Operation $create(O)$, i.e., creating a new object $O \in \mathcal{O}_\mathrm{S}$, is not propagated to $\mathcal{O}_{\mathrm{S}_q^\sigma}$, i.e., it does not impact σ. Similarly, operation $create(m, D, C)$, i.e., forming an association between two objects $D, C \in \mathcal{O}_\mathrm{S}$, does not affect $\mathcal{M}_{\mathrm{S}_q^\sigma}$.[7]

The $delete(O)$ prerequisite ensures that O cannot participate in non-identity morphisms. So, we delete a whole kind with a single property. This has a critical impact on σ if $O \in \mathcal{O}_{\mathrm{S}_q^\sigma}$.

Operation $delete(m)$ is propagated as follows: (i) We try to modify S_q^σ by replacing m with $m' \in \mathcal{M}_\mathrm{S}$, s.t. m and m' form a *comutative diagram* (i.e., the semantics is preserved). (ii) If there is no such m', we remove all unreachable subgraphs of S_q^σ. (iii) The unreachable variables are also removed from S_q^π, filtering, aggregation, and result modifiers. Removing an unreachable variable may cause, e.g., a result containing more records when filtering, a result differently ordered using `ORDER BY`, etc.

Operation $rename(O, n)$ is not propagated to σ as there is no binding between object names and MMQL variable names.

Operation $move(m, p)$ only changes the morphisms, leaving the MMQL variables unchanged. Thus, the move operation has no impact on the evaluability of q except for the clause `DISTINCT` and aggregation functions. In the former case, a different result may be returned. In the latter case, the root of the aggregation may be changed. Nevertheless, the modification of S_q^σ may be significant in the DSL statement, where, e.g., the join or graph traversal may be more complex as we change a base $m \in \mathcal{M}_\mathrm{S}$ to a composite one.

Operations $copy(O)$ and $copy(m, D, C)$ do not impact σ. However, the second one could simplify $\mathcal{M}_{\mathrm{S}_\sigma}$ by replacing a composite morphism with a base morphism m'.

[7]Note that the attributes in the projection of σ must be explicitly specified.

The propagation of $addMapping()/deleteMapping()$ depends on the underlying logical representation of the data. The following complications may arise: (i) We do not change S, but only the mapping. Hence, there should not be any changes in q. But, as depicted in Figure 4, we might need to synchronise the query with the newly defined kinds, as every graph pattern must start in the root of a kind. (ii) MMQL statement does not change, yet target DSL does not allow particular MMQL construct. Then, such construct is evaluated utilising *MM-quecat* [7] as a so-called *postponed statement*, which is not propagated into the target DSL.

5 DEMONSTRATION OUTLINE

In our presentation, we will first demonstrate the described functionality of *MM-evoque* using the example depicted in Figures 2–5. Next, we will demonstrate additional features, such as the representation of the schema category using a sequence of operations or the support for versioning and redundancy.

As the tool is currently a prototype part of a robust research aim of efficient unifying multi-model data management, we will also introduce the broader context and discuss the open problems and challenges.

REFERENCES

[1] Carlo A. Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. 2010. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *Proc. VLDB Endow.* 4, 2 (nov 2010), 117–128.

[2] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive Query Processing. *Found. Trends Databases* 1, 1 (jan 2007), 1–140.

[3] Jérôme Fink, Maxime Gobert, and Anthony Cleve. 2020. Adapting Queries to Database Schema Changes in Hybrid Polystores. In *Proc. of SCAM '20*. IEEE, 127–131.

[4] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2021. Data Dependencies for Query Optimization: A Survey. 31, 1 (jun 2021), 1–22.

[5] Pavel Koupil, Jáchym Bártík, and Irena Holubová. 2022. MM-evocat: A Tool for Modelling and Evolution Management of Multi-Model Data. In *Proc. of CIKM '22*. ACM, 4892–4896.

[6] Pavel Koupil, Daniel Crha, and Irena Holubová. 2023. A Universal Approach for Simplified Redundancy-Aware Cross-Model Querying. *Available at SSRN 4596127* (2023).

[7] Pavel Koupil, Daniel Crha, and Irena Holubová. 2023. MM-quecat: A Tool for Unified Querying of Multi-Model Data. In *Proc. of EDBT '23*. OpenProceedings.org, 831–834.

[8] Pavel Koupil and Irena Holubová. 2022. A Unified Representation and Transformation of Multi-Model Data using Category Theory. *J. Big Data* 9, 1 (2022), 61.

[9] Pavel Koupil, Sebastián Hricko, and Irena Holubová. 2022. MM-infer: A Tool for Inference of Multi-Model Schemas. In *Proc. of EDBT '22*. OpenProceedings.org, 2:566–2:569.

[10] Pavel Koupil, Martin Svoboda, and Irena Holubová. 2021. MM-cat: A Tool for Modeling and Transformation of Multi-Model Data using Category Theory. In *Proc. of MODELS '21*. IEEE, 635–639.

[11] Jiaheng Lu and Irena Holubová. 2019. Multi-model Databases: A New Journey to Handle the Variety of Data. *ACM Comput. Surv.* (2019), 38.

[12] Mark Lukas Möller, Meike Klettke, Andrea Hillenbrand, and Uta Störl. 2019. Query Rewriting for Continuously Evolving NoSQL Databases. In *Proc. ER 2019 (LNCS)*, Vol. 11788. 213–221.

[13] David I Spivak and Ryan Wisnesky. 2015. Relational Foundations for Functorial Data Migration. In *Proc. of DBPL '15*. ACM, 21–28.

[14] Uta Störl and Meike Klettke. 2022. Darwin: A Data Platform for NoSQL Schema Evolution Management and Data Migration. In *Proc. of the EDBT/ICDT '22 Workshops*.