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

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

Jáchym Bártik
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) 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. It extends the function- alities 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, 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 \( C = (O, M, \circ) \) consists of a set of objects \( O \), set of morphisms \( M \), and a composition operation \( \circ \) over the morphisms ensuring transitivity and associativity. Each morphism is modelled as an arrow \( f : A \rightarrow B \), where \( A, B \in O \). \( A = \text{dom}(f) \), \( B = \text{cod}(f) \). And there is an identity morphism \( 1_A \in M \) 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 \( S = (O_S, M_S, o_S) \). Objects in \( O_S \) correspond to the domains of
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 \( M_{\delta} \) 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 \( p: s \rightarrow o \), where \( s \) (e.g., variables or constants) correspond to objects from \( O_{p} \) and \( p \) maps to morphisms in \( M_{p} \). For query \( q \), patterns in selection form a selection category \( S_{p} \) (a subcategory of \( S \)) and patterns in projection form a projection category \( S_{p} \) (a subcategory of \( S_{p} \)).

**Example 3.2.** Figure 5 on the right shows two MMQL queries \( q_{1} \) and \( q_{2} \):

"For each pair of friends, return their names and surnames" and "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 associated with prerequisites that must be fulfilled.

**Table 1: Basic SMOs**

<table>
<thead>
<tr>
<th>SMO</th>
<th>Behaviour</th>
<th>Prerequisite</th>
</tr>
</thead>
<tbody>
<tr>
<td>create((O))</td>
<td>Creates object ( O ) with a unique identifier and default parameters.</td>
<td>( O \in O_{1} )</td>
</tr>
<tr>
<td>create((m, D, C))</td>
<td>Creates morphism ( m:D \rightarrow C ) with a unique signature and default parameters.</td>
<td>( D, C \in O_{1} )</td>
</tr>
<tr>
<td>delete((O))</td>
<td>Deletes object ( O ).</td>
<td>( O \in O_{1} )</td>
</tr>
<tr>
<td>delete((m))</td>
<td>Deletes morphism ( m ).</td>
<td>( m \in M_{1} )</td>
</tr>
<tr>
<td>rename((O, n))</td>
<td>Changes the name of object ( O ) to ( n ).</td>
<td>( O \in O_{1} )</td>
</tr>
<tr>
<td>move((m, p))</td>
<td>Moves morphism ( m ) from ( p ) to ( m ).</td>
<td>( m \in M_{1} )</td>
</tr>
<tr>
<td>copy((O))</td>
<td>Creates a copy ( \mathcal{O}' ) of object ( O ) with a unique signature.</td>
<td>( O \in O_{1} )</td>
</tr>
<tr>
<td>copy((m, D, C))</td>
<td>Creates a copy ( m':D \rightarrow C ) of morphism ( m ) with a new unique signature.</td>
<td>( D, C \in O_{1} )</td>
</tr>
</tbody>
</table>

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.

**Figure 1: A screenshot of MM-evoque**

**Example 4.1.** Consider Figures 2–5 and queries \( q_{1}, q_{2}, q_{3} \) from Example 3.2.

Figure 2 shows a naïve 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
Table 2: Basic SMOs and their impact on MMQL query constructs

<table>
<thead>
<tr>
<th>SMO</th>
<th>create(O)</th>
<th>create(m, D, C)</th>
<th>delete(O)</th>
<th>delete(m)</th>
<th>rename(O, n)</th>
<th>mosw(m, p)</th>
<th>copy(O)</th>
<th>copy(m, D, C)</th>
<th>addMapping()</th>
<th>deleteMapping()</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_{seq}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>\infty</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VALUES</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>\infty</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FILTER</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S_{agg}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LIMIT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Aggregations</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(2) MMQL query \( q \) is translated into a set of constructs. The selection category \( S_{seq} \) and projection category \( S_{agg} \) are created according to respective graph patterns of \( q \).

(3) Using Table 2, we determine whether \( q \) can be synchronised as \( \text{max}(\text{impact} (\text{smo}, \text{construct})) \) 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 \( S_{seq} \) and \( S_{agg} \). 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 MMQL.

Operation \( \text{create}(O) \), i.e., creating a new object \( O \in O_S \), is not propagated to \( O_{agg} \), i.e., it does not impact \( \sigma \). Similarly, operation \( \text{create}(m, D, C) \), i.e., forming an association between two objects \( D, C \in O_S \), does not affect \( M_{agg} \).

The \( \text{delete}(O) \) 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 \( \sigma \) if \( O \in O_{agg} \).

Operation \( \text{delete}(m) \) is propagated as follows: (i) We try to modify \( S_{agg} \) by replacing \( m \) with \( m' \in M_{agg} \), s.t. \( m \) and \( m' \) form a commutative diagram (i.e., the semantics is preserved). (ii) If there is no such \( m' \), we remove all unreachable subgraphs of \( S_{agg} \). (iii) The unreachable variables are also removed from \( S_{agg} \), 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 \( \text{ORDER BY} \) etc.

Operation \( \text{rename}(O, n) \) is not propagated to \( \sigma \) as there is no binding between object names and MMQL variable names.

Operation \( \text{move}(m, p) \) only changes the morphisms, leaving the MMQL variables unchanged. Thus, the move operation has no impact on the evaluability of \( \sigma \) except for the clause \( \text{DISTINCT} \) 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 \( S_{agg} \) may be significant in the DSL statement, where, e.g., the join or graph traversal may be more complex as we change a base \( m \in M_{agg} \) to a composite one.

Operations \( \text{copy}(O) \) and \( \text{copy}(m, D, C) \) do not impact \( \sigma \). However, the second one could simplify \( M_{agg} \), by replacing a composite morphism with a base morphism \( m' \).

The propagation of \( \text{addMapping}() / \text{deleteMapping}() \) 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 \( \text{MM-quecat} \) 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 \( \text{MM- Evoque} \) 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


