

MM-quecat: A Tool for Unified Querying of Multi-Model Data*

Demo Paper

Pavel Koupil Department of Software Engineering, Charles University Prague, Czech Republic pavel.koupil@matfyz.cuni.cz Daniel Crha Department of Software Engineering, Charles University Prague, Czech Republic daniel.crha@gmail.com Irena Holubová Department of Software Engineering, Charles University Prague, Czech Republic irena.holubova@matfyz.cuni.cz

ABSTRACT

The dawn of multi-model data has brought many challenges to most aspects of data management. In addition, no standards exist focusing on how the models should be combined and managed. This paper focuses on the problems related to multi-model querying. We introduce *MM-quecat*, a tool that enables one to query multi-model data regardless of the underlying multi-model database or polystore. Using category theory, we provide a unified abstract representation of multi-model data, which can be viewed as a graph and, thus, queried using a SPARQL-based query language. Moreover, the support for cross-model redundancy enables the choice of the optimal multi-model query strategy.

1 INTRODUCTION

More than 60% of existing most popular database management systems (DBMSs) can be denoted as *multi-model*¹ reflecting the still growing share of multi-model applications.

Example 1.1. An example of a multi-model scenario is shown in Figure 1. The graph data (blue) represents customers, whereas the relational data (purple) is redundant to the graph data. Order details are captured in the document data (green).

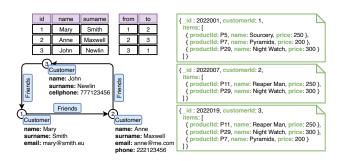


Figure 1: An example of multi-model data

Unfortunately, according to our extensive survey [8], the particular set of models, the way they are combined, or the queries and storage strategies supported vary greatly. This situation is given by the fact that (1) the multi-model DBMSs are based on the different original core single model as well as distinct target application domains, (2) the combined models often have even contradictory features, and (3) there is no acknowledged standard on how to support a combination of models, cross-model querying, multi-model indices, etc.

*Supported by the GAČR projects no. 20-22276S and 23-07781S. ¹https://db-engines.com/en/ranking The described situation brings many new challenges to data management. The primary problem is how to "grasp" all the models and their specifics. For this purpose, we have proposed a so-called *schema category* [5] based on category theory. It enables to view the multi-model data at an abstract level as a *small category* naturally backed by a graph. At the same time, the theory behind it enables us to cover all existing models, their specifics, and types of combinations. Around this representation, we have already built several tools: *MM-cat* [7] for multi-model modeling and transformations, *MM-infer* [6] for inference of a multi-model schema from sample instances, and *MM-evocat* [4] for evolution management of multi-model data.

This paper focuses on the next natural step towards unified management of multi-model data – multi-model querying. As no standards exist, the query languages over multi-model databases also vary greatly [8]. E.g., there are declarative and imperative approaches. Or usually, there exists a (non-standard) extension of SQL or an SQL-like language, but there are also various systemspecific query languages. Another system-specific difference is in the model representing the result – the systems usually only use the original core model.

To address the indicated problems, we introduce a new member of our multi-model family – a tool called MM-quecat² which enables to query the abstract categorical representation and return the result in the abstract categorical model. The main contributions can be summarised as follows:

- The unifying categorical representation enables us to represent all popular data models (relational, key/value, document, column, array, and graph) and all types of their combinations (embedding, references, and redundancy).
- The categorical representation can be viewed as a graph and thus queried using a standard and verified graph query language. For this purpose, we introduce the *Multi-Model Query Language* (MMQL) based on well-known and userfriendly SPARQL [10] notation.
- We target a wide range of users. The SPARQL notation is popular amongst advanced database/web users. Furthermore, we introduce a graphical notation of the query language for less technically skilled users, who can access the data using a manual graph exploration.
- In both cases, the SPARQL query is decomposed and translated to the query languages of the underlying DBMSs, and the intermediate results are then merged to the final result. The categorical representation of the result can then be transformed into any (combination of) models.
- Since the multi-model databases naturally support crossmodel redundancy, there can exist multiple decomposition plans. We enable to view the plans and let users choose their preferred one.

^{© 2023} Copyright held by the owner/author(s). Published in Proceedings of the 26th International Conference on Extending Database Technology (EDBT), 28th March – 31st March, 2023, ISBN 978-3-89318-092-9 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

²https://www.ksi.mff.cuni.cz/~koupil/mm-quecat/ (including a demo video)

• We present a prototype implementation of *MM-quecat* demonstrating the indicated advantages and general user-friendliness of the approach shielding the user from the implementation specifics of particular DBMSs.

Note that the unifying categorical representation does not need to distinguish whether the multiple models reside in a single DBMS or a set of DBMSs (a polystore). Hence, we shield the user from the need to know system-specific query languages, both when migrating data and when using more than one DBMS.

Paper Outline. In Section 2, we review related work. Section 3 introduces the categorical representation of multi-model data. Section 4 describes the presented multi-model querying tool. In Section 5, we outline its demonstration.

2 RELATED WORK

According to our survey [8], the supported query languages in existing multi-model databases vary significantly. The most common approach is an extension of SQL or an SQL-like language. Besides non-standard proposals, such as SQL++ [9], only two ISO standard extensions exist for multi-model querying of relational and document model – SQL/XML [2] and SQL/JSON [3]. However, such an approach requires the user to distinguish the underlying data representation.

As we have mentioned, we base the unification of the multimodel data on its categorical representation, whereas a (small) category can be viewed as a graph. Several approaches already inspired us in this effort, together with the support for their querying. Spivak et al. [11] utilize category theory to represent relational data and their querying using the Categorical Query Language (CQL). It is based on a set of operations with categorical schemas and functors that realize, e.g., join, intersection, or deletion. Similarly, CGOOD [13] utilizes a categorical abstraction of the object-relational data model. The respective query language is based on graph pattern matching. The authors compose composite operations (e.g., projection, join, or difference) from a set of basic operations (e.g., addition and deletion). Similarly, the Algebraic Property Graph [12] (APG), an abstract categorical representation of a property (labeled) graph and RDF data, can be queried using SPARQL. However, all these approaches do not consider multi-model data in its full generality.

Besides our proposal [5], multi-model data is also considered in paper [14], which extends Spivak's approach for relational data towards other models. The authors also propose the usage of complex categorical constructs, namely pullbacks, for joining intermediate results of a complex multi-model query. A tool based on these formal basics, enabling querying of multi-model data using Haskell, is introduced in [15]. However, the complexity of this robust proposal can be seen as the main drawback since the user-friendliness and, thus, expected wide usability is questionable. The redundancy is also not considered.

3 CATEGORICAL CONCEPTUAL MODEL

Let us first remind the basic notions of the 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. Each morphism is modeled as an arrow $f : A \to B$, where $A, B \in O$. We must also ensure *transitivity* ($g \circ f \in M$ for any $f, g \in M, f : A \to B, g : B \to C$), associativity (requiring $h \circ (g \circ f) = (h \circ g) \circ f$ for any suitable $f, g, h \in M$), as well as introduce an *identity* morphism 1_A for each object A such that $f \circ 1_A = f = 1_B \circ f$ for any $f : A \to B$. A category can

be visualized as a multigraph, where objects act as vertices and morphisms as directed edges.

Schema and Instance Category. The core of conceptual modeling of multi-model data in our tools is formed by the schema category. To simplify the understanding, we explain it using the terms known from the ER model, though we do not need to distinguish them, as they are all treated in the same way.

A schema category S is defined as a tuple $(O_S, \mathcal{M}_S, \circ_S)$. Objects in O_S correspond to the ER model's entity types, attributes, and relationship types. Morphisms in \mathcal{M}_S connect appropriate pairs of objects. The explicitly defined morphisms are denoted as *base*, those obtained via the composition \circ as *composite*.

An instance category $I = (O_I, \mathcal{M}_I, \circ_I)$ is defined to represent data instances in a unified way. This category structurally corresponds to the schema category, i.e., a functor $F : I \rightarrow S$ assigns a schema to the data, similar to [13], but it bears the particular data instances.

(For precise formal definitions with technical details see [5].)

Example 3.1. Figure 2 depicts the schema category of sample data from Figure 1 and the instance category of its respective part.

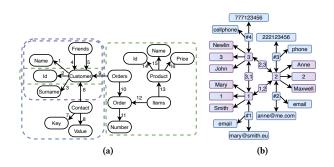


Figure 2: An example of schema category (a) and instance category (b) of data from Figure 1

Mapping. The decomposition of a schema category S, eventually partial or overlapping, is defined via a set of *mappings*, as also formally defined in [5] and omitted for paper length. Each mapping describes where and how data instances of a subgraph of S (following specific conditions) are stored in a selected single-/multi-model DBMS.

Example 3.2. The colors in Figure 2 depict the decomposition into distinct models indicated in Figure 1.

Since the terminology within the particular popular models differs, we provide the unification of respective model-specific terms in [7]. For instance, a *kind* corresponds to a class of items represented in each model, e.g., a relational table or a collection of JSON documents. For each kind, the mapping specifies the respective DBMS, its name, its root object in S, and an *access path* which recursively describes the structure of a kind, i.e., its (simple or complex) properties, relatively to the root object. The description is rich enough to cover various specifics of the underlying models and their combinations, such as properties with user-defined, anonymous, or dynamically-derived names; properties *inlined* from more distant parts of the categorical graph (via composite morphisms); auxiliary properties used, e.g., for logical grouping of a set of properties; order-preserving/orderignoring sets of sub-properties, etc.

4 QUERY ENGINE MM-QUECAT

To enable a robust and user-friendly way of querying over the categorical graph, we have implemented a prototype tool called *MM-quecat*. It is an extensible modular framework that belongs to our family of tools built on top of the categorical representation of multi-model data. Currently it supports the following models and DBMSs: *PostgreSQL*³ (relational and document, i.e., multi-model), *Neo4j*⁴ (graph), *MongoDB*⁵ (NoSQL document), and *Apache Cassandra*⁶ (columnar model). For the purpose of expressing the query, we proposed a modification of SPARQL, i.e., a popular language for querying graph data, called MMQL (see Section 4.1). To extend the target group of users, we also introduce its graphical expression/visualization.

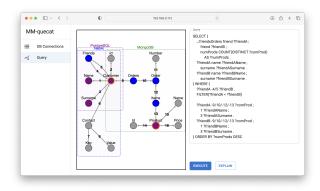


Figure 3: A screenshot of MM-quecat

In Figure 3 we provide a screenshot of *MM-quecat*. On the left, we can see a similar categorical model and decomposition as in Figure 2. Note that for simplicity, we define the decomposition at the level of DBMSs, which can be single- or multi-model. On the right, there is an MMQL query accessing the data. The colors of the nodes represent the objects used in the query (blue), projections (purple), aggregation (red), and filtering (orange).

In general, *MM-quecat* evaluates a query naturally: The specified query is first mapped to the schema category, and, depending on its decomposition, it is decomposed into particular *query parts*. The query parts are then translated into the *Domain Specific Languages* (DSLs) and evaluated in the respective DBMSs. The returned *intermediate results* are then transformed into the categorical representation and merged into the resulting schema category. Eventually, the remaining part of the query evaluation is applied to the merged data. Depending on the user requirements, it can be transformed into the selected (multi-model) representation.

Example 4.1. Let us consider the following multi-model query over the schema in Figure 2 and data in Figure 1: "*For each pair of friends, count the number of products ordered by both friends. Sort the results in descending order according to the number of products.*"

Figure 4 depicts the workflow of evaluation of the query. In Phase I. the user specifies the MMQL query over the schema category. The body of the SELECT clause describes the hierarchical structure of the result, where we want to return the customer's first and last name and the number of common products ordered by both customers. The WHERE clause describes the graph pattern that we match to the data. We also have a filtering condition in clause FILTER to eliminate duplicate pairs of friends in swapped order, we apply aggregation COUNT on DISTINCT pairs, and we sort them using clause ORDER BY.

In Phase II., the query is decomposed into two query parts: the graph (blue) model of Neo4j and the document (green) model of MongoDB. In Phase III. the queries in the respective DSLs are provided, namely the blue query for Neo4j and the green query for MongoDB. Phase IV. involves evaluation of the queries and gaining the results. In Phase V. we can see their representation as the instance categories whereas the query is finalized, i.e., the data is merged, and the remaining query operations are applied in Phase VI. In Phase VII. we can see its transformation to the relational model, i.e., neither of the original models is used.

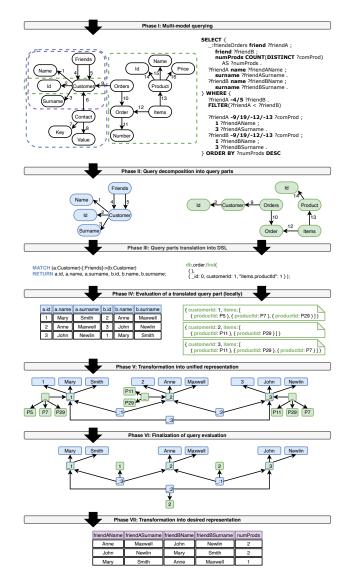


Figure 4: A workflow of query evaluation

As we can see, some phases of the evaluation process are performed with an external tool. Namely, the evaluation of query parts is ensured natively by the respective single/multi-model DBMSs. Also, the process of transforming data from and to the categorical representation is ensured by the multi-model transformation library of *MM-cat* [7]. The core issue of the process and the main contribution of *MM-quecat* is the decomposition of the query into query parts, the translation of MMQL to DSLs, and the merging of the intermediate results together with the

³https://www.postgresql.org/

⁴https://neo4j.com/

⁵https://www.mongodb.com/

 $^{^{6}} https://cassandra.apache.org/_/index.html$

eventual final evaluation of the remaining parts of the query. The translation process corresponds to the mapping of constructs of MMQL depicted in Table 1. Query decomposition reflects the decomposition of the schema category. Inspired by [1, 13], for merging and unified processing of the intermediate results, we utilize categorical constructs, such as product (Cartesian product), pullback (join), coproduct (disjoint union), or pushout (union), to be formally described in a separate paper.

4.1 Multi-Model Query Language (MMQL)

As we have mentioned, the multi-model query language supported in *MM-quecat*, called MMQL, directly re-uses the syntax and semantics of SPARQL (for example, see Figure 4, Phase I) with the following modifications: Instead of RDF graph data we query the schema category, so, e.g., we do not use IRIs, but we query over schema category, and the result is represented as an instance category. For this purpose, we modify clause SELECT inspired by clause CONSTRUCT. The other clauses' semantics remain unchanged (respecting the different domains).

The particular constructs supported in MMQL are listed in Table 1 together with the comparison with the constructs commonly used in particular models and their database representatives. (Due to space limitations, we only compare with the currently supported database representatives.)

Table 1: Comparison of constructs supported in MMQL and in single-model query languages

MMQL	PostgreSQL (SQL)	Neo4j (Cypher)	MongoDB	Cassandra (CQL)
FROM	FROM	-	db.collection	FROM
SELECT	SELECT	RETURN	\$project	SELECT
WHERE	WHERE	WHERE	\$match	WHERE
FILTER	condition(s)	condition(s)	condition(s)	condition(s)
COUNT, MIN, MAX, AVG	GROUP BY HAVING	COUNT, MIN, MAX, AVG	aggregate()	GROUP BY
graph pattern	JOIN	MATCH	\$lookup	-
OPTIONAL	OUTER JOIN	OPTIONAL MATCH	-	-
UNION	UNION	UNION	\$unionWith	-
ORDER BY	ORDER BY	ORDER BY	sort	ORDER BY
OFFSET	OFFSET	SKIP	skip	-
LIMIT	LIMIT	LIMIT	limit	LIMIT
DISTINCT	DISTINCT	DISTINCT	distinct	DISTINCT
AS	AS	AS	"alias" : "\$field"	AS
{ SELECT }	(SELECT)	CALL MATCH	-	-

4.2 Redundancy and Query Plans

Another essential feature of the proposed categorical representation and thus all our related tools, including *MM-quecat* is the support for redundancy. For more efficient query evaluation (a part of) the data can be stored in multiple models, corresponding, e.g., to classical materialized views. Consequently, the query can be evaluated in different ways, i.e., using different evaluation plans. *MM-quecat* supports this feature at the multi-model level, i.e., it detects all query decompositions into query parts. Then it evaluates the cost of such decomposition using a combination of evaluation of the query parts and their merging. We assume that either the underlying DBMS or a system-specific wrapper provides such a cost to enable the estimation for a particular query plan. The user can then analyze the query plans, including their costs. *Example 4.2.* In Figure 5 we provide an alternative evaluation of the graph (blue) query part from Figure 4 Phase II using the relational (purple) model. Its cost is higher due to the need to evaluate two joins.



Figure 5: An alternative query part and its translation

5 DEMONSTRATION OUTLINE

In our demonstration, we will first briefly show the whole context of *MM-quecat*, i.e., how the user can create a schema category. Having explained the unifying graph representation, we will demonstrate the usage of MMQL, i.e., the range of constructs that can be used and the way they are translated into DSLs, i.e., the query plans. For this purpose, we will create a representative set of queries inspired by the multi-model benchmark *UniBench* [16]. Finally, we will demonstrate the support for multi-model redundancy, i.e., different query plans.

As the tool is currently a prototype part of a robust research aim targeting self-adaptation of multi-model databases with regards to the efficiency of query evaluation, we will discuss not only the advantages of the unifying categorical representation but also the open problems and challenges of multi-model querying.

REFERENCES

- Kristopher Brown, David Spivak, and Ryan Wisnesky. 2019. Categorical data integration for computational science. *Computational Materials Science* 164 (2019), 127–132.
- [2] ISO. 2011. ISO/IEC 9075-14:2011 Information technology Database languages – SQL – Part 14: XML-Related Specifications (SQL/XML).
- [3] ISO. 2017. ISO/IEC TR 19075-6:2017 Information technology Database languages – SQL – Part 6: SQL support for JavaScript Object Notation (JSON).
- [4] 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.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] Jiaheng Lu and Irena Holubová. 2019. Multi-model Databases: A New Journey to Handle the Variety of Data. ACM Comput. Surv. (2019), 38.
- [9] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. The SQL++ Query Language: Configurable, Unifying and Semi-structured. https: //doi.org/10.48550/ARXIV.1405.3631
- [10] Eric Prud'hommeaux and Andy Seaborne. 2008. SPARQL Query Language for RDF. W3C. http://www.w3.org/TR/rdf-sparql-query/.
- [11] Patrick Schultz, David I. Spivak, Christina Vasilakopoulou, and Ryan Wisnesky. 2017. Algebraic Databases. *Theory & Applications of Categories* 32, 16-19 (2017), 547–619.
- [12] Joshua Shinavier, Ryan Wisnesky, and Joshua G. Meyers. 2019. Algebraic Property Graphs. https://doi.org/10.48550/ARXIV.1909.04881
- [13] Chris Tuijn and Marc Gyssens. 1996. CGOOD, a Categorical Graph-oriented Object Data Model. *Theoretical Computer Science* 160, 1 (1996), 217–239.
- [14] Valter Uotila and Jiaheng Lu. 2022. A Formal Category Theoretical Framework for Multi-model Data Transformations. *CoRR* abs/2201.04905 (2022). arXiv:2201.04905
- [15] Valter Uotila, Jiaheng Lu, Dieter Gawlick, Zhen Hua Liu, Souripriya Das, and Gregory Pogossiants. 2021. MultiCategory: Multi-model Query Processing Meets Category Theory and Functional Programming. *Proc. VLDB Endow.* 14, 12 (2021), 2663–2666.
- [16] Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. 2018. UniBench: A Benchmark for Multi-Model Database Management Systems. In TPCTC.