Reshaping Reality: Creating Multi-Model Data and Queries from Real-World Inputs

Author1	Author2	Author3
Institution	Institution	Institution
Country	Country	Country
email1	email2	email3

Abstract

The *variety* characteristic of Big Data introduces significant challenges for verified single-model data management solutions. The central issue lies in managing the *multi-model data*. As more solutions appear, especially in the database world, the need to benchmark and compare them rises. Unfortunately, there is a lack of available real-world multi-model datasets, the number of multi-model benchmarks is still small, and their general usability is limited.

This paper proposes a solution that enables the creating of multi-model data from virtually any given single-model dataset. We propose a pipeline of a subset of our tools for multi-model data management that enables automatic inference of a schema, its user-defined modification, and the data generation reflecting the changes. Using the well-known Yelp dataset, we show its advantages and usability in three scenarios reflecting reality.

Multi-model data, data transformation, real-world datasets

1 Introduction

Although the traditional relational data model has been the preferred choice for data representation for decades, the advent of Big Data has exposed its limitations in various aspects. Many technologies and approaches considered mature and sufficiently robust have reached their limits when applied to Big Data. One of the most daunting challenges is the *variety* of data, which encompasses multiple types and formats originating from diverse sources and inherently adhering to different models. There are structured, semi-structured, and unstructured formats; order-preserving and order-ignorant models; aggregate-ignorant and aggregate-oriented systems; models where data normalization is critical or the redundancy is naturally supported; etc.

The naturally contradictory features of the so-called *multi-model data* introduce an additional dimension of complexity to all data management aspects, including modelling, storing, querying, transforming, integrating, updating, indexing and many more. Hence, several multi-model tools for data management have emerged. For example, considering storing multi-model data, more than 2/3 of the 50 most widely used

database management systems (DBMSs)[1] now fall under the category of *multi-model* following the Gartner prediction [11] made almost 10 years ago. Unfortunately, no standards exist on which models to combine and how, so each DBMS provides a proprietary solution.

Similarly, there exist *polystores* [19, 9], sometimes denoted as *multi-database systems*. The general idea is that several distinct data management systems (usually single-model) live under a common, integrated schema provided to the user. Polystores can be further classified [23] depending on various aspects, such as the number of query interfaces or the types of underlying systems (homogeneous or heterogeneous), the level of autonomy of the underlying systems, etc. So, again, the variety of choices is wide.

Also, the number of query languages reflects the situation of the existing systems when considering the multi-model data. Despite being primarily denoted for the relational model, numerous SQL-like languages are currently used for various other models and their combinations. For instance, the Cassandra Query Language (CQL)[2] is an SQL-like query language in column-family DBMS Apache Cassandra. Or, the Array Query Language (AQL)[3], also SQL-like, is used in array DBMS SciDB to query multi-dimensional arrays. And SQL has also been extended to support cross-model querying. Apart from numerous proprietary solutions [18], there are also two ISO standards for querying relational and document data – SQL/XML [15] and SQL/JSON [16]. In general, an extensive overview of existing multi-model query languages (developed within various DBMSs) is provided in a recent survey [13].

Considering the range of each area's approaches, choosing the optimal tool for the particular use case is highly challenging. Naturally, we need to be able to compare the selected set of tools on *all* target use cases, and benchmarking comes into play. Despite many single-model benchmarks and data generators for all the common models (see Section 2), the shift to the multi-model world is not straightforward. The multi-model test cases must cover the required subset of models and their mutual relations, such as multi-model embedding, cross-model references, or multi-model redundancy. In addition, the variety of use cases grows with the number of distinct models combined. Hence, the number of truly multi-model benchmarks is small, and their versatility and coverage are limited.

In response to this problem, we propose a solution that enables the creation of virtually any possible multi-model data set together with the respective operations. To ensure the data sets have realistic characteristics, we do not utilize the classical approach of exploitation of generators, providing values with a required distribution. Instead, we expect to input a real-world single-model data set. Our approach is based on utilizing the toolset we have developed in our research group for various aspects of multi-model data management based on the unifying categorical representation of multi-model data – the so-called *schema category* [1]. This abstract graph representation backed by the formalism of category theory enabled us to build a family of tools for modeling and transformations [2], schema inference [3], data migration under evolving schema [4], querying using SPARQL-like query language MMQL [5], or query rewriting [6]. In

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

[2] `https://cassandra.apache.org/doc/stable/cassandra/cql/`

[3] `https://www.nersc.gov/assets/Uploads/scidb-userguide-12.3.pdf`

this paper, we show that by using an appropriate composition of the tools (and adding an adequate interface), we can create a tool for transforming single-model data and queries to any possible combination of multi-model data and queries for benchmarking.

Outline In Section 2 we overview related work. In Section 3, we introduce the categorical representation of multi-model data and the tools we utilize in the proposal. In Section 4, we introduce the multi-model transformation pipeline and provide an illustrative example using Yelp dataset. In Section 5, we conclude and outline future steps.

2 Related Work

Two main obvious approaches to benchmark data management tools exist. We can use existing, preferably real-world datasets or a data generator that outputs synthetic, pseudo-realistic datasets. Although we can find many representatives of both, most focus on a single selected model. The number of multi-model representatives is very low.

2.1 Repositories

Considering the well-known repositories of real-world datasets, the most popular model is relational, reflecting the history and popularity of relational DBMSs. The second most popular model is hierarchical, expressed usually in JSON [14], the main format supported in NoSQL document DBMSs. There are also repositories of graph data, as this model represents specific use cases, hardly captured by the previous two.

The most popular repositories are usually related to research activities. There are general repositories such as the Kaggle repository[4] of datasets for data science competitions (involving, e.g., Titanic survival data), the UCI Machine Learning Repository[5] for machine learning research (involving, e.g., census data), the IEEE DataPort[6], or the Harvard Dataverse[7]. The open-access repository Zenodo[8], developed under the European OpenAIRE program, enables researchers to share datasets and other research outputs. For graph data, there are popular repositories such as the Stanford Large Network Dataset Collection[9], the Network Data Repository[10], or the Open Graph Benchmark[11].

The open data movement naturally provides another good source of data. Many

[4] https://www.kaggle.com/
[5] https://archive.ics.uci.edu/
[6] https://ieee-dataport.org/
[7] https://dataverse.harvard.edu/
[8] https://zenodo.org/
[9] https://snap.stanford.edu/data/
[10] https://networkrepository.com/
[11] https://ogb.stanford.edu/

governments (e.g., US[12], UK[13], EU[14], etc.) provide open data portals hosting various datasets on demographics, economics, transportation, and public health. Similarly, Amazon Web Services (AWS) host a variety of open datasets[15] that can be accessed and analyzed directly in the cloud.

Various datasets can be found also in GitHub[16], or related projects, such as DataHub[17]. Or, one can search the whole Internet, e.g., using the Google Dataset Search[18].

2.2 Generators

Often, we cannot easily find a suitable real-world dataset. In that case, we can use a data generator or a comprehensive benchmark with a data generator capable of producing pseudo-realistic datasets with required natural features (e.g., distribution of values or structural features). However, to our knowledge, most existing generators are limited to a single, specific data model or format, or they are constrained to a fixed set of one or a few use cases, each represented by a dataset and related operations. For example, popular benchmarks, such as TPC-H and TPC-DS[19], are naturally focused on the relational data model. Similarly, benchmarks like XMark [21] or DeepBench [8] are tailored to the document data model involving basic NoSQL or path-finding queries. A comprehensive review of purely graph data generators is presented in [10]. Or for instance, GenBase [22] focuses on the array data model and queries for array manipulation.

Considering multi-model data, only a few representatives fall into this category. BigBench [12] covers semi-structured and unstructured data and the relational data model, but it lacks support for both graph and array data models. UniBench [24] does not support the array data model either, and it considers only a single use case within the benchmark. Finally, M2Bench [17] encompasses relational, document, graph, and array data models. Nevertheless, despite each covered benchmark task involving at least two data models, the benchmark is designed to fit within one of three predefined use cases.

3 Categorical View and Management of Multi-Model Data

Multi-model data refers to data represented by multiple interconnected logical models within a single system. The interconnection can be done in several ways:

1. The two (or more) models can be mutually *embedded*. For example, a JSONB

[12] https://data.gov/
[13] https://www.data.gov.uk/
[14] https://data.europa.eu/
[15] https://registry.opendata.aws/
[16] https://github.com/
[17] https://datahub.io/
[18] https://datasetsearch.research.google.com/
[19] https://www.tpc.org/

column in PostgreSQL[20] enables embedding a JSON document into a relational table.

2. A *reference* can exist between two entities residing in different modes.

3. The same part of data can be represented *redundantly* using multiple models.

Integrating different data models within a larger system, such as a polystore or a multi-model DBMS, allows for using the most appropriate model for specific tasks. For example, structured data with slight variations might best suit the document model. Data with numerous relationships requiring efficient path queries may fit the graph model. Or, rapidly generated data with simple querying needs could be handled by the key/value model.

3.1 Categorical Representation of Multi-Model Data

To grasp the popular models' specific features, we have first proposed the so-called *schema category* [1], a unifying abstract categorical representation of multi-model data to manage any possible combination of known models.

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

The *schema category* is then defined as a tuple $\mathbf{S} = (O_\mathbf{S}, \mathcal{M}_\mathbf{S}, \circ_\mathbf{S})$. Each schema object $o \in O_\mathbf{S}$ is internally represented as a tuple (*key*, *label*, *superid*, *ids*), where *key* is an automatically assigned internal identity, *label* is an optional user-defined name, $superid \neq \emptyset$ is a set of attributes (each corresponding to a signature of a morphism) forming the actual data contents a given object is expected to have, and $ids \subseteq \mathcal{P}(superid)$, $ids \neq \emptyset$ is a set of particular identifiers (each modeled as a set of attributes) allowing us to distinguish individual data instances uniquely. Each morphism $m \in \mathcal{M}_\mathbf{S}$ is represented as a tuple (*signature*, *dom*, *cod*, *label*). The explicitly defined morphisms are denoted as *base*, obtained via the composition $\circ_\mathbf{S}$ as *composite*. The *signature* allows us to distinguish all morphisms except the identity ones mutually. For base morphism, we use a single integer number. For composite morphism, we use the concatenation of signatures of respective base morphism using the $\cdot$ operation. *dom* and *cod* represent the domain and codomain of the morphism. Finally, *label* $\in \{$ `#property`, `#role`, `#isa`, `#ident` $\}$ allows us to further distinguish morphisms with semantics "has a property", "has an identifier", "has a role", or "is a". (We provide explanatory examples in Section 4).

To unify the terminology from different models, we use the following terms [1]: A *kind* corresponds to a class of items (e.g., a relational table or a collection of JSON documents) and a *record* corresponds to one item of a kind (e.g., a table row or a JSON document). A record consists of simple or complex *properties* having their *domains*.

[20] `https://www.postgresql.org/`

3.2 Categorical Multi-Model Data-Management Toolset

The schema category (together with its mapping to the underlying models) allows us to seamlessly handle any combination of models and process them independently of the system. When a specific operation needs to be performed at this abstract level, it is passed down to the underlying database system for execution.

During the last couple of years, our research group has developed a family of tools that enable one to manage multi-model data represented using category theory. The tools we will utilize for our proposed purpose are the following:

- M [2] enables multi-model modeling, i.e., the manual creation of the schema category representing the conceptual model and mapping to any combination of the logical models of supported DBMSs. It also supports the respective data transformations between the logical and categorical models.
- E [4] extends M with the propagation of changes in the categorical schema to data instances.
- I [3] enables (semi-)automatic inference of the schema category from sample multi-model data instances.
- Q [7] adds support for querying over the schema category using the *Multi-Model Query Language* (MMQL) [5] based on well-known SPARQL [20] notation. The queries are then decomposed according to the mapping to logical models. The subqueries are evaluated in the underlying DBMSs, and the partial results (if any) are combined to produce the final result.
- P [6] extends Q with the propagation of changes in the categorical schema to queries in MMQL according to the mapping.

4 Multi-Model Transformation Pipeline

Although the original aim of the listed tools is different, if we join them together in a specific way, which is possible thanks to the common categorical representation of multi-model data, we can gain a pipeline that enables a user-friendly and efficient way to generate pseudo-realistic multi-model data. On the input, we assume a real-world single-model data set (or, eventually, a synthetic one with reasonable characteristics or a multi-model dataset we want to modify). On the output, we want to get multi-model data created from the input data based on user requirements. Eventually, the users can also provide a query over the input data, and we want to output its respective modification reflecting the data transformation (if it exists). We can identify several scenarios where such a pipeline is applicable:

- **Scenario A:** The users provide input data with model X, and they want to transform it to model X'.
- **Scenario B:** The users provide input data having model X, and they want to transform its part to model X' and the rest to X'', whereas a multi-model DBMS that supports both X' and X'' exists.

- **Scenario C:** The users provide input data having model X, and they want to transform its part to model X' and the rest to X'', whereas none of the DBMSs we consider supports both X' and X''. So, the data is stored in two DBMSs.

Our pipeline covers all three scenarios. To explain the ideas, we first provide a running example based on a subset of the Yelp Open Dataset[21]. The data describes Yelp's businesses, reviews, and user data, all represented using the JSON format.

Fig. 1 involves a part of the input dataset. We can see JSON document collections *User*, *Review*, *Checkin*, *Business*, and *Tip*, i.e., the data represented in the original JSON document model (green). Next to the documents, we can see the initial schema category inferred by [redacted]. The green nodes represent the roots of the respective kinds. In the compound brackets, we can see the identifiers of the kinds (e.g., the property *review_id* for kind *Review*, or the pair of properties *user_id*, *business_id* for kind *Tip*). The arrows represent morphisms – in this simple example, only the most common type "has a property" (whose label we omit for simplicity), i.e., leading to simple/complex properties of the kinds.

As we can see, the quality of the initial schema category is limited by the input data quality, the input model's specific features, and the capabilities of [redacted]. In particular, the properties denoted with red color bear values of identifiers of various kinds, as they probably represent the respective references. E.g., kind *Review* is identified by *review_id*, but it also involves *user_id* of the user who created the review and *business_id* of the reviewed business. This cannot be captured using JSON, but we want to capture this information in the schema category and use it later. Similarly, kind *User* has a set of properties (denoted with pink color) that have the same (in this case simple) structure and semantics (as we can guess from their names *compliment_**) and differ only in type. And there might be lots of such properties. So, at the categorical (conceptual) level, expressing them as a single property with a particular type might make more sense and can be represented better in another logical model.

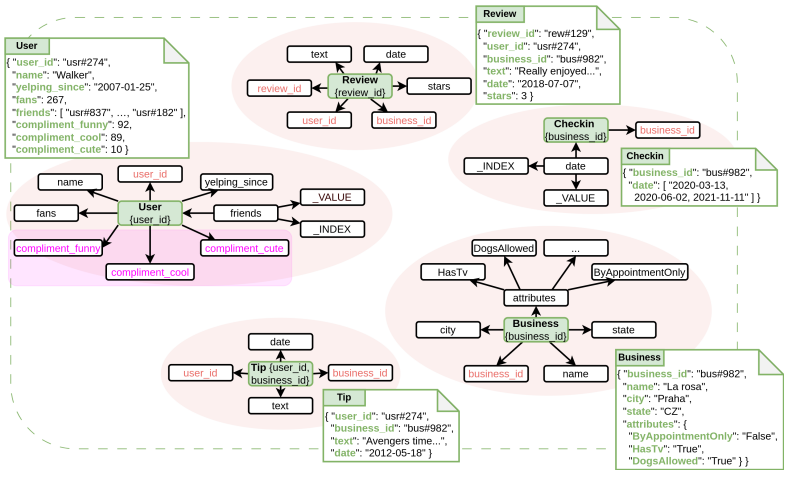

Figure 1: Input single-model JSON document collections and inferred initial schema category

[21] https://www.yelp.com/dataset

Fig. 2 depicts the situation after the users visualize the initial schema category in M and edit it using its extension E to solve the issues.[22] So first, the users replaced repeating occurrences of properties *business_id* and *user_id* and expressed the references using the morphisms with the respective direction (the new morphisms are emphasized with dotted arrows). Second, the properties of kind *User* that are structurally and semantically equivalent were merged and transformed into a single property with a respective property *_TYPE*.

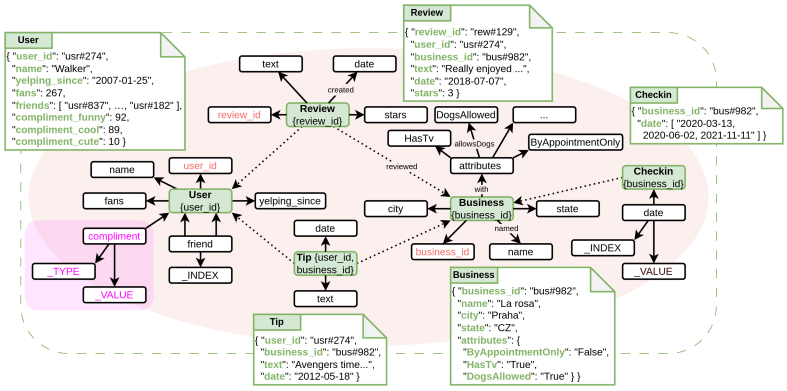

Figure 2: Edited (improved) schema category from Fig. 1

Having the edited schema category, we can use E again to modify the mapping (initially to the input document model). Following **scenario A**, we want to transform all the JSON document data into the relational model. The situation is depicted in Fig. 3, where the user changes the mapping of the whole schema category to the relational model (violet). Namely, the original kinds *User*, *Review*, *Business*, and *Tip* are mapped to respective relational tables instead of JSON collections. Regarding the features of the relational models, also property *friend* of kind *User* and property *date* of kind *Checkin* had to be mapped to separate kinds *Friend* and *Date* and thus respective separate relational tables. Similarly, the property *attribute* of kind *Business* was mapped to a map of attributes and thus a separate table.

Following **scenario B**, the users might find out that transforming all the data to the relational model is not optimal, and they decide to use the best of both worlds. As depicted in Fig. 4, they keep the mapping of kinds *User* and *Friend* to the relational models, each to a separate table, like in Fig. 3. They also want to keep a mapping of kinds *Review* and *Tip* to the relational model but to merge them into a single table because *Tip* is just a subset of *Review*. So, they create a new kind *Comment* that covers both of them and map it to a single table. The new kind requires property *comment_id*, which we can reuse (for records of kind *Review*) or generate by a simple algorithm (for records of kind *Tip*).

Finally, they decided to embed the kinds *Attribute* and *Date*, which required separate relational tables, to the relational table of kind *Business*. So they mapped them to the document model and embedded them to the kind *Business*. (Such a combination of models is supported, e.g., in PostgreSQL.) This transformation reduces the overhead of joining the same tables each time while keeping the kind *Business* mapped to the relational model.

Fig. 4 depicts the result, where we get truly multi-model data represented in two logical models – violet relational and green document.

[22] Some issues can be solved in I (semi-)automatically, we use them just for illustration.

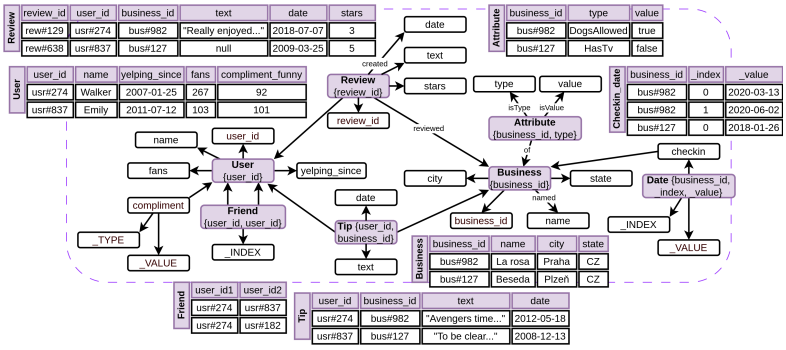

Figure 3: Schema category from Fig. 2 mapped to the relational model (scenario A)

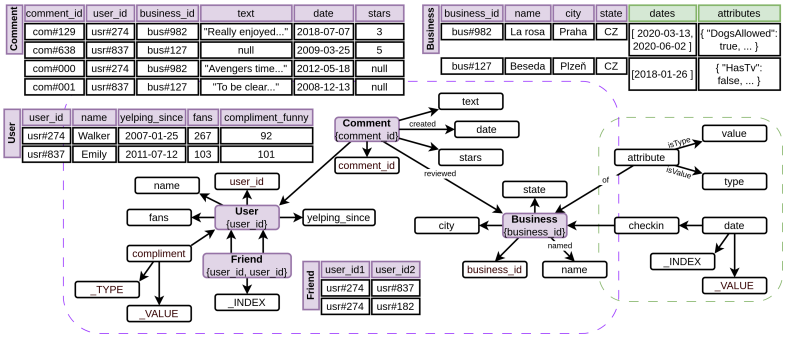

Figure 4: Schema category from Fig. 2 mapped to relational and document model (scenario B)

Finally, following **scenario C** and as depicted in Fig. 5, the users might further transform the multi-model data from a combination of two to a combination of three logical models and map the kind *Comment* to the wide-column model (red). This model is better suited for frequent data analysis, i.e., the type of queries the users might want to do with the comments. It also more naturally represents that tips do not have all the attributes of reviews.

So, as we can see, using the pipeline, it is very simple to transform the input data to any multi-model data only by modification of the schema category and its mapping to the logical models. Nevertheless, we may also want a similar functionality for the queries. Extending the pipeline with Q makes it possible to query over the schema category using MMQL [5], a graph query language inspired by SPARQL. Depending on the specified mapping of the schema category to the logical models, Q translates the MMQL query to be evaluated in the underlying DBMS. But, for our purposes, instead of querying, we only retrieve the query with the transformed data and use it for benchmarking.

For example, the users may want to query for "*names of businesses which have been reviewed since January 1st, 2023 and allow dogs*". Its expression in MMQL over the improved schema category in Fig. 2 is provided in Fig. 6. If the input data in Fig. 2 were stored in Mon-

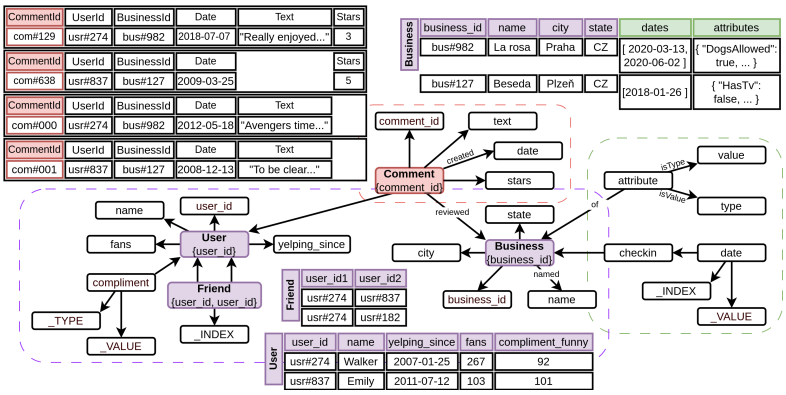

Figure 5: Schema category from Fig. 4 mapped to relational, document, and wide-column model (scenario C)

goDB[23], its translation to MongoDB QL is provided in Fig. 7.

```
SELECT {
    ?business name ?name .
}
WHERE {
    ?business -reviewed/created ?date ;
        with/allowsDogs "true" ;
        named ?name .
    FILTER(?date > "2023-01-01")
}
```

Figure 6: MMQL query over the improved schema category in Fig. 2

If we change the mapping to another model (or a combination models) represented in another DBMS (or multiple DBMSs), we get the query expressed using the respective query language(s). In addition, if we change the part of the schema category accessed by the query, P ensures the modification of MMQL, together with the mapping.

When we unify the business attributes to a map, as depicted in Fig. 3, the MMQL query is modified to reflect the change, as depicted in Fig. 8. In addition, in Fig. 3, we also changed the mapping to the relational model (**scenario A**). Assuming that now the data is stored in PostgreSQL, the respective mapping to the relational model ensures the translation of MMQL query to the SQL query provided in Fig. 9.

If we use the combination of the document and relational model (**scenario B**) depicted in Fig. 4, we can assume that the data is still stored in PostgreSQL. As SQL in PostgreSQL is extended towards the support of cross-model queries over both relational and document data, i.e., SQL/JSON, the evaluation process again translates the MMQL query to a single, this time cross-model query, as depicted in Fig. 10. Note that despite the mapping change, the parts of the

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

```
db.review.aggregate([
    { $match: { date: { $gt: ISODate('2023-01-01') } } },
    { $lookup: {
        from: "business",
        localField: "business_id",
        foreignField: "business_id",
        as: "business"
    } },
    { $match: { attributes: { DogsAllowed: true } } },
    { $project: {
        _id: 0,
        name: "$business.name"
    } },
])
```

Figure 7: MongoDB QL query over data from Fig. 2

```
SELECT {
    ?business name ?name .
}
WHERE {
    ?business -reviewed/created ?date ;
        -of ?attribute ;
        named ?name .
    ?attribute isType "DogsAllowed" ;
        isValue "true" .
    FILTER(?date > "2023-01-01")
}
```

Figure 8: MMQL query over schema category from Fig. 3

```
SELECT business.name AS name
FROM business
JOIN review ON business.business_id = review.business_id
JOIN attribute ON business.business_id = attribute.business_id
WHERE review.date > '2023-01-01'
    AND attribute.type = 'DogsAllowed'
    AND attribute.value = true
```

Figure 9: SQL query over data from Fig. 3 (scenario A)

schema category accessed by the MMQL query remain untouched, so the MMQL query remains the same.

Finally, suppose we use a combinati on of models unsupported by a single multi-model DBMS (**scenario C**) depicted in Fig. 5. In that case, the evaluation consists of the decomposition of the query to two subqueries for the respective subsystems – SQL for PostgreSQL and, e.g., CQL for Apache Cassandra[24] – as depicted in Fig. 11. Thus, we can also test a family of DBMSs, together with the need to use an additional tool to merge the results. However, because

[24] https://cassandra.apache.org/

```
SELECT business.name AS name
FROM business
JOIN comment ON business.business_id = comment.business_id
JOIN attribute ON business.business_id = attribute.business_id
WHERE comment.date > '2023-01-01'
    AND attributes->>'DogsAllowed' = 'true'
```

Figure 10: SQL/JSON query over data from Fig. 4 (scenario B)

the schema category did not change, the MMQL query stays the same again.

```
SELECT business_id
FROM comment
WHERE date > '2023-01-01'
```

```
SELECT name AS name
FROM business
WHERE business_id IN (/* CQL query result */)
    AND attributes->>'DogsAllowed' = 'true'
```

Figure 11: CQL and SQL queries over data from Fig. 5 (scenario C)

4.1 Architecture

Fig. 12 provides the schema of the architecture of the proposed pipeline. The expected work with the pipeline is as follows:

1. The users provide the input single-model data to be transformed. The data can be stored in one of the supported DBMSs or provided in files.
2. I parses the data and infers a schema that the schema conversion module transforms to the initial schema category. The schema category is visualized using M.
3. Using its extension E, the users can modify it depending on their requirements. The users can only change the mapping of the schema category to selected combinations of logical models, or they can also change the schema category itself. E transforms the data according to the given mapping.
4. In addition, using Q, the users can specify an MMQL query, which is then automatically translated according to the given mapping. Eventually, the MMQL query may need to be updated using P to reflect the changes in the schema category.

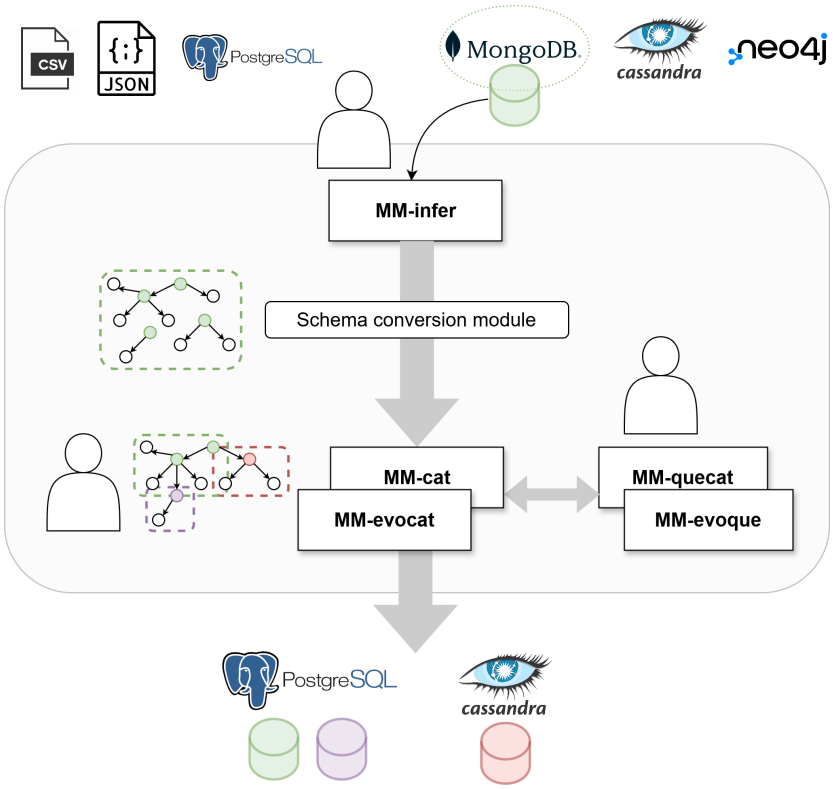

Figure 12: Architecture of the pipeline

4.2 Evaluation of the Proposed Solution

Table 1 provides an overview of the advantages of pipeline utilization compared to manual data/query transformation. On average, depending on the complexity of the data, it is much faster. The pipeline enables us to infer the initial schema category and, thus, get the overall view of the data structure quickly. Also, all special cases and outliers are immediately provided to the users in a visual form. Also, the specification of the requested output is fast, and the transformation is performed automatically without the need to know the specific features of the underlying systems.

Of course, we assume the pipeline supports all the required systems for which we want to create the testing data. However, integrating a new DBMS is simple, as it only requires implementing a respective wrapper. Once we have it, we do not need to implement any transformation script, and we can express the modification only by interacting with the pipeline tools. Consequently, we avoid numerous user-defined errors, as the users are shielded from the technical details. Thus, we do not require an expert familiar with the specifics of various DBMSs.

The flexibility of the pipeline compared to manual data transformation is not limited. As mentioned above, although the pipeline currently supports MongoDB, Post-

greSQL, neo4j, Apache Cassandra, JSON files, or CSV files, new DBMSs and data formats can be easily added using wrappers.

Table 1: Comparison of approaches using different metrics

Metric	**Without pipeline**	**Using pipeline**
Time Required (hours)	10+ (estimation)	0.5 (estimation)
Lines of Code	200+	0
Potential for Errors	High (coding, manual transformation)	Low (tool has been tested)
User Expertise Required	Advanced	Beginner / Intermediate
Flexibility / Customization	High	High

Finally, Table 2 illustrates the time required for user interactions across scenarios A, B, and C depicted in Figs. 3, 4, and 5 when processing the Yelp dataset, comparing the conventional manual approach versus using the proposed pipeline tool. The pipeline streamlines the workflow by automating every step of the process. In Step 1, it infers the schema from the data. Following this, the pipeline facilitates editing the schema in Step 2 by providing a user-friendly interface that allows users to make necessary adjustments with minimal effort. In Step 3, it supports creating custom mappings between different data models. It then moves on to generate multi-model data in Step 4. Finally, it translates queries to operate across different data models in Step 5. The results demonstrate a substantial reduction in the time required for each step when using the pipeline, highlighting its efficiency and effectiveness in reducing user input and eventual errors.

Table 2: User interaction needed in particular scenarios for the Yelp dataset

Scenario	**Step**	**Without pipeline (min)**	**Using pipeline (min)**	**Difference (min)**
A	Step 1	120	5	+115
	Step 2	120	10	+110
	Step 3	120	12	+108
	Step 4	180	2	+180
	Step 5	180	0	+180
B	Step 3	180	12	+168
	Step 4	240	2	+238
	Step 5	240	0	+240
C	Step 3	240	16	+234
	Step 4	300	2	+298
	Step 5	300	0	+300

5 Conclusion

This paper proposes a solution to the problem of lack of real-world multi-model data (and the respective queries). We use a different approach instead of the common strat-

egy of generating a synthetic dataset despite having numerous realistic features. Using a specific utilization of our previously created toolset, we introduce the idea of a transformation pipeline that can transform a given, preferably real-world, dataset into a preferred multi-model dataset. Using a well-known dataset Yelp, we demonstrate the advantages and applicability of the idea.

Our future work will focus primarily on implementing a common interface that will cover the whole functionality of the proposed pipeline and simplify the integration of the tools. In addition, we want to focus on the simulation of the evolution of the resulting datasets, either through user specification or through the detection of changes in the input single-model data or operations. Lastly, we want to create a repository of the resulting multi-model datasets to provide a robust source of test cases to be immediately used. We also want to perform extensive experiments with these datasets to provide unbiased benchmarking results for elected multi-model databases.

Acknowledgment

This work was supported by...

References

[1] Author. Title. *Journal*, Volume(Number):Pages, Year.

[2] Author. Title. In *Booktitle*, page Pages, Address, Year. Publisher.

[3] Author. Title. In *Booktitle*, page Pages, Address, Year. Publisher.

[4] Author. Title. In *Booktitle*, page Pages, Address, Year. Publisher.

[5] Author. Title. *Journal*, Volume(Number):Pages, Year.

[6] Author. Title. In *Booktitle*, page Pages, Address, Year. Publisher.

[7] Author. Title. In *Booktitle*, page Pages, Address, Year. Publisher.

[8] Stefano Belloni, Daniel Ritter, Marco Schröder, and Nils Rörup. DeepBench: Benchmarking JSON Document Stores. In *Proceedings of the 2022 Workshop on 9th International Workshop of Testing Database Systems*, DBTest '22, page 1–9, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Carlyna Bondiombouy and Patrick Valduriez. Query processing in multistore systems: an overview. *Int. J. Cloud Comput.*, 5(4):309–346, 2016.

[10] Angela Bonifati, Irena Holubová, Arnau Prat-Pérez, and Sherif Sakr. Graph Generators: State of the Art and Open Challenges. *ACM Comput. Surv.*, 53(2), apr 2020.

[11] Donald Feinberg, Merv Adrian, Nick Heudecker, Adam M. Ronthal, and Terilyn Palanca. Gartner Magic Quadrant for Operational Database Management Systems, 12 October 2015, 12 October 2015.

[12] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. BigBench: towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1197–1208, New York, NY, USA, 2013. Association for Computing Machinery.

[13] Qingsong Guo, Chao Zhang, Shuxun Zhang, and Jiaheng Lu. Multi-model query languages: taming the variety of big data. *Distributed and Parallel Databases*, May 2023. Publisher Copyright: © 2023, The Author(s).

[14] Ecma International. JavaScript Object Notation (JSON), 2013. `http://www.JSON.org/`.

[15] ISO. ISO/IEC 9075-14:2011 Information technology – Database languages – SQL – Part 14: XML-Related Specifications (SQL/XML), 2011.

[16] ISO. ISO/IEC TR 19075-6:2017 Information technology — Database languages — SQL — Part 6: SQL support for JavaScript Object Notation (JSON), 2017.

[17] Bogyeong Kim, Kyoseung Koo, Undraa Enkhbat, Sohyun Kim, Juhun Kim, and Bongki Moon. M2Bench: A Database Benchmark for Multi-Model Analytic Workloads. *Proc. VLDB Endow.*, 16(4):747–759, dec 2022.

[18] Jiaheng Lu and Irena Holubová. Multi-Model Databases: A New Journey to Handle the Variety of Data. *ACM Comput. Surv.*, 52(3), June 2019.

[19] Jiaheng Lu, Irena Holubová, and Bogdan Cautis. Multi-model Databases and Tightly Integrated Polystores: Current Practices, Comparisons, and Open Challenges. In *Proc. of CIKM 2018*, pages 2301–2302, Torino, Italy, 2018. ACM.

[20] Eric Prud'hommeaux and Andy Seaborne. *SPARQL Query Language for RDF*. W3C, January 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[21] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: a benchmark for XML data management. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, page 974–985. VLDB Endowment, 2002.

[22] Rebecca Taft, Manasi Vartak, Nadathur Rajagopalan Satish, Narayanan Sundaram, Samuel Madden, and Michael Stonebraker. GenBase: a complex analytics genomics benchmark. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 177–188, New York, NY, USA, 2014. Association for Computing Machinery.

[23] Ran Tan, Rada Chirkova, Vijay Gadepally, and Timothy G. Mattson. Enabling query processing across heterogeneous data models: A survey. In *BigData*, pages 3211–3220, 2017.

[24] Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. UniBench: A Benchmark for Multi-model Database Management Systems. In *TPCTC 2018*, pages 7–23, Cham, 2019. Springer International Publishing.