Multi-Model Database Management Systems - a Look Forward

Zhen Hua Liu [1] , Jiaheng Lu[2], Dieter Gawlick[1], Heli Helskyaho[2,3]

Gregory Pogossiants[4],   Zhe Wu[1]

[1]Oracle Corporation [2]University of Helsinki [3]Miracle Finland Oy [4]Soulmates.ai

Abstract. The existence of the variety of data models and their associated data processing technologies make data management extremely complex. In this paper, we envision a single Multi-Model DataBase Management Systems (MMDBMS) providing declarative accesses to a variety of data models. We briefly review the history of the evolution of the DBMS technology to derive requirements of MMDBMSs and then we illustrate our ideas of building MMDBMSs satisfying those requirements. Since the relational algebra is not powerful enough to provide a mathematical foundation for MMDBMSs, we promote the category theory as a new theoretical foundation, which is a generalization of the set theory. We also suggest a set of shared data infrastructure services among data models to support "Just-In-Time" multi-model data access autonomously.

1 INTRODUCTION – why MMDBMS?

Here is a short history of databases: Initially, database management systems supported the hierarchical and the network model (e.g., IBM's IMS and GE's IDS respectively). These databases evolved very fast and developed the core infrastructures, such as journaling, transactions, locking, 2PC (group and fast commit), recovery, restart, fault tolerance, high performance, TP-monitors, messaging, main storage databases, and much, much more. We still use these concepts today. In the 80' and 90', these databases were widely replaced by the relational database management systems (RDBMS). The main argument is its solid theoretical foundation: set based relational data model and declarative query language (SQL) based on abstract algebra over set processing.

However, the demands to simplify the interaction between applications and databases with simple storage and querying interfaces are not always possible using only the relational model. Object databases ODBMS (Object Database Management Systems) filled this gap by providing easy access to objects with object-oriented programming languages. With additional OO features in RDBMSs, ORDBMSs are able to support many domain data types, such as text, spatial, and images data. Interestingly, the last decade has witnessed the re-emergence of hierarchical data models in the form of XML and JSON data and the re-emergence of the network data model in the form of RDF semantic graph and property graph data. This has led to native XML, JSON, graph database systems and ORDBMSs providing XML, JSON, RDF and graph data support via SQL/XML, SQL/JSON standards and ongoing standard development to provide graph access via SQL. More applications are adopting graph modelling and graph query since graphs provide a flexible way to structure application data and adapt them dynamically to changes [15, 27, 29]. The source of graph data could come from relational, XML, or JSON that exist in the different databases.

The history of the database evolution has shown that new applications often require new data models leading to extended infrastructure of DBMSs with new query languages over these new data models. One existing solution is the polyglot persistency approach, which leverages numerous DBMSs to support different data models and integrates them programmatically at the application layer. The biggest issue of polyglot persistency is that the combined DBMSs is neither declarative nor unified. It leaves database application to procedurally join data among multiple data models and manually transform among data model instances. Instead of putting the burden on applications, it is more desirable to have a unified single DBMS, which hides the complexity of multiple data models by providing declarative approach of querying multi-model data instances and just-in-time data model transformation.

In this position paper, we advocate a multi-model database management system (MMDBMS) that has the ability to incorporate any data model and allows users to manipulate all data models declaratively. Users are able to explore the real power of an MMDBMS by leveraging its ability to autonomously transform data from one data model to another. MMDBMSs allow data providers and data consumers to look at the same data using different models depending on their most effective view. MMDBMSs accomplish these data model transformation autonomously on behalf of users.

We argue that the design of a full-fledged MMDBMS requires a more powerful mathematical foundation. The last few decades have witnessed a tremendous success of RDBMSs leveraging the relational algebra as theoretical foundation and therefore limiting this foundation to relational data. We recognize the same data can be represented relationally, hierarchically, graphically and are thus queryable by SQL, XQuery, Property-Graph Query Language respectively. Therefore, we feel the need of having a new theoretical foundation to provide transparent data model and query language transformations among those data models and languages. In other words, MMDBMSs require a powerful mathematical foundation to reason about declarative data model transformation among multiple data models. In this paper, we promote category theory [5,14] shall be able to play the role of the new mathematical foundation to reason declarative construction and transformations among various data models.

In addition, this paper describes a set of shared data infrastructure services. The shared services not only include essential common data services, such as transaction, recovery, security, high availability but also include integrating artificial intelligence to provide “Just-In-Time” data model access and telemetry service to promote multi-model situation awareness service [4].

Organization The remainder of this paper is structured as follows: Section 2 introduces the preliminaries on categories and examples of model transformation. Section 3 presents category theory as the mathematical foundation for MMDBMS. Section 4 illustrates MMDBMS infrastructure services. Section 5 shows related work and section 6 concludes the paper.

2. Preliminaries on categories and model transformation

The category theory exists since 1940 and has been successfully used in many mathematical, physical and computer science areas. Recently, researchers have applied it to the databases area (e.g. the functorial query language in [9,10,11]). In this section, we review the concept of *category*, *functor* and give an example to perform cross-model transformation between relation and JSON data.

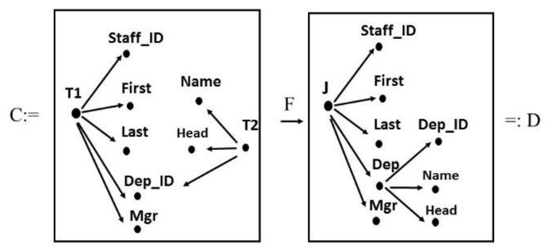
Definition 1[19]. *A **category** consists of a collection of objects, a collection of morphisms, so that:*

- *Each morphism has specified domain and codomain objects; the notation $f : X \to Y$ signifies that f is a morphism with domain X and codomain Y.*
- *Each object has a designated identity morphism: $X \to X$.*
- *For any pair of morphisms f, g, there exists a composite morphism whose domain is equal to the domain of f and whose codomain is equal to the codomain of g.*

Definition 2[19]. *A **functor** F: C → D, between categories C and D, consists of the following data:*

- *An object $F_C \in D$, for each object $c \in C$,*
- *A morphism F: $F_C \to F_{C'} \in D$, for each morphism f: $c \to c' \in C$, so that the domain and codomain of F are, respectively equal to F applied to the domain or codomain of f.*

Each category can be considered as a collection of objects with some relations between them, expressed by "*morphisms*" – special form of describing relation dependency of the objects. One category could be mapped to some other by "*functors*". Mapping category *C* to *D* means mapping objects in such a way that relationship between mapped objects in *D* will be inducted by the corresponded relation in *C*. With category theory, MMDBMSs can be considered as a container that hosts multiple data sets of different data models as multiple different categories.



(a) The mapping between relational categories and JSON category

Employee (T1)				
Staff_ID	First	Last	Dep_ID	Mgr
100	John	Smith	D1	Null
101	James	William	D2	100

Department (T2)		
Dep_ID	Name	Head
D2	Sales	101
D3	Production	102

(b) Two example tables with data instances

Fig. 1. A functor F: *C* → *D* from the schema of relation to that of JSON

In particular, a functor F: C → D of database schemas is a mapping that takes vertices in C to vertices in D and arrows in C to arrows or paths (a sequence of arrows) in *D*. For example, in Figure 1, each of the six leaf vertices: *Staff_ID*, *First*, *Last*, *Dep_ID*, *Name*, *Head* in *C* is mapped to the vertex in *D* of the same label. This mapping is not necessarily bijective (e.g. the vertex of *Dept* in *D* cannot map to any vertex in *C*). Based on this mapping, we will discuss in turn three functors on the level of data instances in Example 1-3, which transform JSON instances to relational instances and vice versa, illustrated in Table 1.

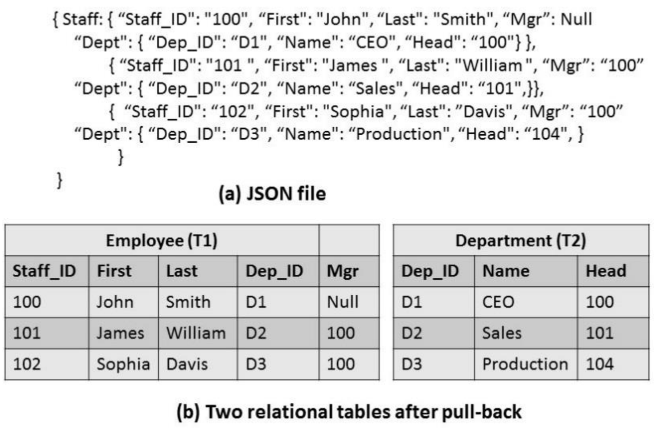
Category operations	Symbol	Database operations
Pullback	Δ_F: *J-inst* → *R-inst*	JSON to relation
Right Pushforward	$\prod_F$: *R-inst* → *J-inst*	Relation to JSON by inner-join
Left Pushforward	$\sum_F$: *R-inst* → *J-inst*	Relation to JSON by outer-join

Table 1. Illustration of data instance transformations induced by three functors.

*Example 1 (**Pullback**) We explore the pullback functor Δ_F by applying it to a JSON file depicted in Figure 2. This operation splits the JSON document into two tables.*

In the next two examples, we will explore the right and left pushforward functors induced by Figure 1.

*Example 2 (**Right Pushforward**) We explore the right pushforward functors $\prod_F$ by applying on two tables in Figure 1(b). The JSON file is described in Figure 2. The JSON file can be considered as the inner-join of two relational tables.*



```
{ Staff: { “Staff_ID": "100", “First": "John", “Last": "Smith", “Mgr”: Null
   “Dept": { “Dep_ID": “D1", “Name": “CEO", “Head": “100"} },
         { “Staff_ID": "101 ", “First": "James ", “Last": "William ", “Mgr”: “100”
   “Dept": { “Dep_ID": “D2", “Name": “Sales", “Head": “101",}},
         {  “Staff_ID": “102", “First": "Sophia", “Last": ”Davis", “Mgr”: “100”
   “Dept": { “Dep_ID": “D3", “Name": “Production", “Head": “104", }
          }
}
```

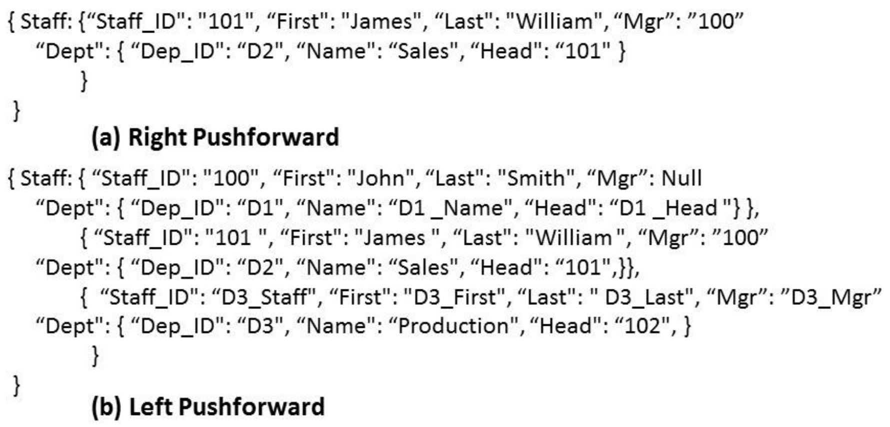
(a) JSON file

Employee (T1)				
Staff_ID	First	Last	Dep_ID	Mgr
100	John	Smith	D1	Null
101	James	William	D2	100
102	Sophia	Davis	D3	100

Department (T2)		
Dep_ID	Name	Head
D1	CEO	100
D2	Sales	101
D3	Production	104

(b) Two relational tables after pull-back

Fig. 2. Example JSON file and the tables after pull-back operations



```
{ Staff: {“Staff_ID": "101", “First": "James", “Last": "William", “Mgr”: ”100”
   “Dept": { “Dep_ID": “D2", “Name": “Sales", “Head": “101" }
         }
}
```

(a) Right Pushforward

```
{ Staff: { “Staff_ID": "100", “First": "John", “Last": "Smith", “Mgr”: Null
   “Dept": { “Dep_ID": “D1", “Name": “D1 _Name", “Head": “D1 _Head "} },
         { “Staff_ID": "101 ", “First": "James ", “Last": "William ", “Mgr”: ”100”
   “Dept": { “Dep_ID": “D2", “Name": “Sales", “Head": “101",}},
         {  “Staff_ID": “D3_Staff", “First": "D3_First", “Last": " D3_Last", “Mgr”: ”D3_Mgr”
   “Dept": { “Dep_ID": “D3", “Name": “Production", “Head": “102", }
          }
}
```

(b) Left Pushforward

Fig. 3. JSON files after Right Pushforward and Left Pushforward functors

*Example 3 (**Left Pushforward**) In this example, we explore the left pushforward functor. Instead of being an inner-join, as in the case of above, the JSON file is formed by the union of two tables. In order to deal with the fact that the record do not have department information, the respective value are skolemized. In other words, the cell is simply added by a new “variable” .*

As shown in the above example, based on category schema and functor mapping, a relational model can be created on top of a hierarchical XML or JSON object. Conversely, a hierarchical model can be created on top of relational rows to access relational object. Therefore, category theory builds the mathematical foundation for the transformation of data instances between different models, which will be further elaborated in the following sections.

3 MMDBMS Framework and Category Theory

Building RDBMSs based on the abstract/set algebra as theoretical foundation has been a tremendous success. The ORDBMS technology is a subsequent successful engineering framework to enable RDBMS to accommodate object data. However, an ORDBMS by itself is not a MMDBMS since MMDBMSs require an improved mathematical foundation to reason about a declarative data model transformation among multiple data models, something not covered by the relational or object-relational algebra. To facilitate transformations among multiple data models, we argue that the category theory [5,14,19] is a more appropriate theoretical foundation for MMDBMS. In this section, we will discuss query transformation, view processing, in-memory processing in MMDBMSs and their potential connections to the category theory.

3.1 Query transformation

MMDBMS enables a query over one data model to be transparently rewritten into an equivalent query over another data model. For example, given the property graph defined in Figure 4, one can run various graph queries (e.g. shortest path). We investigate the following two graph queries (GQ1 and GQ2) in the context of MMDBMS.

Description	Graph query (PGQL v1.1)	Equivalent SQL query
Find a department D1's head and all individuals who report directly or indirectly to D1's head	*GQ1: PATH fp AS ()-[e:Mgr]-> () SELECT emp MATCH (emp)-/:fp+/->(h)-[e1:Head]->(d WITH Dep_ID="D1")*	*WITH g(Staff_ID, First, Last, Dep_ID, Mgr, Depth) AS (SELECT Staff_ID, First, Last, Dep_ID, Mgr, 1 as Depth FROM T1 UNION all SELECT T1.Staff_ID, T1.First, T1.Last, T1.Dep_ID, g.Mgr, 1+ g.Depth as Depth FROM g, T1 where T1.Mgr = g.Staff_ID) SELECT g.Staff_ID FROM g WHERE Mgr is not null and Mgr in (SELECT Head FROM T2 WHERE T2.Dep_ID='D1')*
Find a complete list of departments that an employee John and his whole team belong to	*GQ2: PATH fp AS ()-[e:Mgr]-> () SELECT DISTINCT d MATCH (d) <-[e1:Dept]-(emp) -/:fp+/-> (j WITH First="John")*	*WITH g(Staff_ID, First, Last, Dep_ID, Mgr, 1 as Depth) AS (FROM Staff_ID, First, Last, Dep_ID, Mgr, 1 + g.Depth as Depth FROM T1 UNION all FROM T1.Staff_ID, T1.First, T1.Last, T1.Dep_ID, g.Mgr, g.Depth as Depth from g, T1 where T1.Mgr = g.Staff_ID) SELECT DISTINCT Dep_ID FROM g WHERE Mgr = (SELECT T1.Staff_ID FROM T1 WHERE T1.First='John')*

Table 2. Graph and Relational Model Transformations

Assume the graph data stays in the underlying relational base tables T1, T2, shown in Figure 1. The two declarative graph PGQL queries (Property Graph Query Language, see specifications in [22]) in Table 2 are created on top of T1 and T2. Table 2 also shows equivalent SQL implementations using a recursive WITH clause.

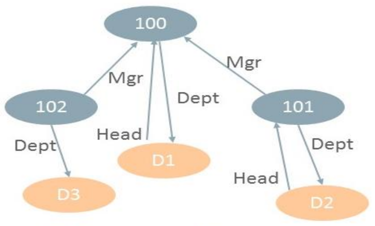


Fig 4. An example of property graph

Category theory enables query rewriting crossing different data models. In particular, we consider each model data-set of a MMDBMS as a category and declare one or many functors to transform objects from one category to another through query languages. Further, a natural transformation provides a way of transforming one functor into another while respecting the internal structure (i.e., the composition of morphisms) of the categories involved. Hence, a natural transformation can be considered a “morphism of functors”. Sometimes two quite different constructions yield the “same” result; this is expressed by a natural isomorphism between the two functors. Therefore, the transformation between functors can be employed to investigate the equivalence between queries with different models.

3.2 Multi-Model Data View Processing

A view is nothing but a query. However, a view bridges the gap between how data is logically organized from the perspective of users and how the data is physically organized from the perspective of the underlying DB system. In MMDBMSs, a view is also a simple way to virtually access one data model on top of another via declarative data model transformation. **This is the power of MMDBMSs, which allows an application to store data in one data model and later query the same data by another application using a different data model via defining multi-model data views**. Thus, multi-model data view realizes the genuine value of MMDBMS which enables data applications requiring different data models to share the same MMDBMS and rely on MMDBMS to transparently provide different access views of the same data adaptive to each data application's requirement.

For example, JSON documents and XML documents are based on hierarchical data models. However, JSON and XML documents can be viewed relationally by decomposing the hierarchy into set of rows and columns. This is achieved by defining XMLTABLE() or JSON_TABLE() view on top of the native storage of XML or JSON documents to view the XML and JSON content relationally. Similarly, contents in a set of relational tables can be weaved constructively to create hierarchical model through SQL/XML and SQL/JSON generation function. Therefore XML view or JSON view can be achieved by defining SQL/XML and SQL/JSON generation function on top of the relational tables. In general, when transformation functions are defined to convert data between different data models, views can be defined to use the transformation function to present desired data model even though the underlying storage of the data may not be in that data model physically. Using relational tables and JSON files in Fig 1 and Fig 2, Table 3 shows SQL/JSON views for constructing relational data model from JSON data model and construct JSON data model from relational model.

Table 3. JSON and Relational Model Transformations

Description	Query

Q1- Construct JSON View from relational content	*CREATE JSON_VIEW AS*
	SELECT JSON {"Staff" : {"STAFF_ID" : e.staff_id, "First" : e.first, "Last" : e.last, "Mgr" : e.mgr, {"Dept" : {"Dept_ID" :d.dept_id, "Names" : d.name, "Head" : d.head }} }
	FROM Employee e, department d
	WHERE e.dep_id = d.dep_id

Q2 - Construct relational view of employee from JSON	*CREATE EMPLOYEE_REL_VIEW AS*
	*SELECT **
	FROM JSON_VIEW f, JSON_TABLE (f.Staff COLUMNS (Staff_ID, First, Last, Mgr)

Q3 - Construct relational view of department from JSON file	*CREATE DEPARTMENT_REL_VIEW AS SELECT * FROM JSON_VIEW f, JSON_TABLE(f.Dept COLUMNS (Dep_ID, Name, Head)*

One approach to improve query over such multi-model view is to through query rewrite technique by leveraging the underlying algebraic property of the transformation functions among different data models. For example, relational predicates over XMLTABLE() or JSON_TABLE() views can be written into XPATH/JSON PATH predicates directly navigating the native hierarchical storage of the underlying XML or JSON data[2]. Inversely, XPATH/JSON path query over the views constructed via SQL/XML or SQL/JSON generation functions over relational tables can be written into relational predicate over the underlying relational table storage [2,13]. Table 4 shows applying query rewrite transformation for Q2 and Q3 of Table 3. The intermediate JSON view don't need to be physically materialized.

Description	Query
Q2	*SELECT Staff_id, First, Last, Mgr FROM Employee*
Q3	*SELECT Dep_ID, Name, Head FROM Department*

Table 4. Query Rewrite Transformation Results

Providing view update capability is feasible provided that transformation function is reversible. The MMDBMS can manage a set of built-in transformation functions between different models and understand the reversibility of the transformation function so that it can automatically determine if the view is updatable. For views that are not updatable systematically due to lack of inverse function, instead-of-update trigger can be supplied by users to deal with ad-hoc update. Therefore, the category theory can be used to reason about view updatability when a natural transformation is reversible. An MMDBMS optimizer manages a set of transformation algebraic rules between different models for query optimization and rewrite and understand the reversibility of the algebraic transformations for view update feasibility.

3.3 Multi-Model In-memory Processing

The classical RDBMS technology assumes the data layout on disk is the same as the one cached in the buffer cache. However, the use of columnar main-memory structures has revolutionized this model [3]. By decoupling the storage format from their in-memory data format, MMDBMSs shall decide how to provide fast in-memory access for multi-model data. There is probably no single best storage format for a data model instance to satisfy all the workload requirements. Using functors and natural transformation in the category theory, MMDBMSs may autonomously rearrange and cache data in a different format as compared to the format stored on disk. MMDBMSs need not to lock down an optimized universal way of storing multi-model data and yet being able to provide ‘Just-In-Time’ data model access through materializing data model instances in alternative forms in NVRAM or RAM.

When the query over multi-model view is complex enough so that it is not feasible to do such query rewrite, for example, performing arbitrary directional path navigations in a hierarchical tree model, then materialized multi-model view technique can be used. The materialized view performs the actual transformation of the data into the physical model to run the query. In classical RDDBMS, materialized view has to be persistent, an in-memory only materialized view is a good option for MMDBMS to have because in-memory materialized view can be populated to speed up query without incurring the overhead of persistency management. Furthermore, the in-memory transformation can be implemented in background without delaying or blocking the foreground DML operations on the original base data. This is in the same way as how in-memory columnar population of the on-disk row format serves a good alternative to persistently migrating from row storage format to columnar storage format [6]. Rather than attempting to determine a best storage format of data upfront to satisfy potentially all workload, in-memory materialized view provides a flexible mechanism to decouple the storage format of data in view from the base table. In this way, neither the system nor the users need to lock down one way of storing and indexing the data. Applying this idea to RDBMS, users may have columnar physical storage with row oriented views materialized in memory or may have row oriented physical storage with columnar view materialized in memory.

In summary, MMDBMS shall use meta-algebra among data model instances to decide whether and when to materialize data model instances physically or in memory or to leave it as a logical entity to avoid materialization cost while providing good access performance. This is the key to integrate in-memory query processing with MMDBMS.

3.4 Multi-Model Security Access via view

Just as classical RDBMS, view serves as an effective mechanism to enforce data security. In classical RDBMS, security can be enforced at table level, row level, column level via column projection and row filtering criteria declared declaratively by users. In MMDBMS, we shall be able to leverage view to enforce data security. Although MMDBMS has concept of table as collection and documents in a collection as a row so that inter-document security can be enforced just as relational DB case, the intra-document security enforcement is not a clear cut as the document does not have concept of column. However, we shall still be able to leverage the concept of view to enforce the intra-document security. For example, a view using XPATH to project a set of XML fragments within an XML document can be used to define part of an XML document that access is granted or revoked. Users may not be granted any privilege to access the base table or collection, however, they are granted access to views defined using document filter and projection to define fine-grained security access privilege.

3.5 Multi-Model with flexible schema

In MMDBMS that aligns with the idea of data first/schema later or never, there is another creative usage of view in MMDBMS. It allows system and users to use adaptive schema concept [1] to access the data. The key to provide adaptive schema access is to leverage view. For example, in a heterogeneous collection of semi-structured documents, the underlying semi-structured data have loose structures so that defining a full schema over the semi-structured data could result in a very large sparse schema with many choices and uncertainties. Such full schema may not be practically useful for query analysis. Therefore, multiple views with various degrees of exposure of the underlying semi-structure data can be provided to users for their query use cases. In this way, users and their applications are not required to lock down one schema of the data using classical E-R design resulting a set of physically materialized tables but rather may have multiple logical flexible schemas, each of which represents certain way of viewing and accessing the data [23]. The underlying data may be stored natively in user input format as the source of the truth. There are many views defined in the system. Some of the views are virtual that require on-the-fly query rewrite to push the access to the underlying storage data. Other views are materialized persistently or computed and populated on-demand in-memory to speed up query access. The advantage of supporting multi-view approaches is that there are no schema evolution and physical data migration issues that classical RDBMS with fixed schema has. This is because there is only one source of truth using the original data directly from the import of the user without using any physical schema to shred the original data. All views are secondary whose content are always re-computable from the original physical data based on the schema that the system and user agree upon [23]. Through automatic schema derivation efforts by the MMDBMS system, users have tremendous flexibility to pick and choose views needed for their current applications and are able to keep evolving their view definition without physically migrating the storage data.

3.6 Application of Inverted index in MMDBMS

Borrowed from the idea from IR inverted index on full text search, MMDBMS may extend the inverted index to a universal index that indexes not only full text content, but also other multi-model data instances and their schema. The inverted index in MMDBMS is model-context-aware schema and data search. The advantage of such inverted search index is that users do not need to know what to index in advance yet still enjoy high performance for an explorative type of queries. For example, searching keywords from inverted index shows the occurrence of the keywords under different data models: keywords within XML documents or JSON objects under a hierarchical path, within graph structures inside a graph traversal path, or within relational rows under a specific set of columns. From the search result of a multi-model context aware inverted index, users can then use model domain specific index to further narrow down the search and query criteria.

The existing ORDBMS supports domain index as a way to index domain specific data. MMDBMS inherits the domain index approach. However, a new usage of domain index is that it can be used as a secondary index after general inverted index. Furthermore, just as in RDBMS, conventional B+ tree index scan might be slower than in-memory columnar scan for an unselective query. Similar investigation needs to be carried out to evaluate if in-memory scan in MMDBMS is compatible with adaptive domain index for different kinds of multi-model data.

3.7 Benefits of MMDBMS Users

We summarize two main benefits for MMDBMS users:

∗ **Flexible data models.** In a MMDBMS, there is no primary data model. It does not matter that the data are initially defined with hierarchical data model or relational model or graph model etc. It will be easy for users to start the operation with one model and later add and incorporate new data models as their use cases demand. Users are able to incorporate multiple data models into a single MMDBMS and manage them in a holistic way.

∗**Transparent cross-model query transformation and rewriting**. All query languages are equal in MMDBMSs. Each data model may have its own domain-specific declarative query and modification language. User applications may initially start with one data model language to access one data model. However, as they start to manage multiple data models and try to transform, join and mix data models in MMDBMSs, MMDBMSs understand the connections among these data models and their respective languages so that it can transparently do query transformation and rewriting among different data model languages on behalf of applications. The genuine value of MMDBMSs is that they allow many data applications to share one DBMS and on-demand declaratively transform from one data model to another. It is the MMDBMS's' (not users') responsibility to optimize and execute inter-data model queries and modification requests.

3.8 Limitation of Category Theory

Some logical limitation of Category Theory has been pointed by Jean–Yves Girard [24], renowned logician in Proof Theory area. Girard considered the notion of equivalence in Category Theory to be too strict "up to isomorphism". Therefore, he wrote, the Category Theory can't operate with other, more complicated equality forms, in-particular in novel logic theory [25, 26]. However, right now we are far away from such problems in our MMDBMS approach.

4 Infrastructure Services – Ecosystem for MMDBMS

MMDBMSs need to provide a set of common services that can be used by the different models. This section describes some of these key services:

∗ **Atomicity:** Databases allow users to bundle requests; this is the well-known transaction support. The fundamental idea is that if a transaction is unable to complete there will not be any trace of that transaction in the database and if it succeeds, everything is committed permanently. To achieve this any required database object has to be isolated from other transactions progressing in parallel and changes are not visible until a transaction completes successfully. However, this basic (ACID) model has significant limitations in respect to functionality, performance and scalability. A well-known technology is to use escrow technology, which supports parallel updates for commutative operations that are commonly frequently available for inventory management. Additionally, weaker models exist for those applications that can tolerate relaxed support; e.g., BASE. MMDBMSs need to support a wide range of models and therefore support different levels of atomicity for different data models and different use cases. The level of atomicity should be able to be tuned differently for different data models and use cases based on their requirements both manually and automatically.

∗ **Fault resilience for recovery and provenance:** Fault resilience is a service to capture information to recover from any error and avoid any loss of data while supporting atomicity. Implementing fault resilience for an MMDBMS is challenging but it should be done better than with a polyglot solution due to its single integrated backend. The requirements for fault resilience can be different for diverse data models and use cases. It should be able to be tuned both manually and automatically.

∗ **Telemetry:** This service will provide a base for a wide range of data capturing and analysis services in MMDBMS. The basic usage is to understand the system behavior for a wide range of perspectives; e.g., understanding and debugging functionality and performance, understanding usage patterns, and billing. A more advanced usage is to identify abnormal behavior in real time. Multi-model data can be analyzed in real time and/or externalized for provenance and offline-analysis. For this service, MMDBMSs should be able to be more adaptive and flexible, such as adaptive in-memory processing, adaptive universal multi-model indexing, and adaptive view processing via query rewrite or in-memory materialization and adaptive schema view for just-in-time semi-structured data in-memory processing.

∗ **Machine Learning:** MMDBMSs will use machine learning on top of telemetry Data to understand user's data application so that it can recommend suitable data model for applications. As applications evolve different data model might become appropriate. MMDBMSs will autonomously build the appropriate data model just-in-time to better serve users. At a more advanced level, telemetry and machine learning can also identify abnormal behavior such as faulty sensors, as well as assets and system components.

5 Related Work

A multi-model data management system is designed to address the variety challenge of a complex world. In general, there exist two solutions: (i) polyglot persistence and (ii) multi-model database.

The history of polyglot persistence may trace back to the federation of relational engines or distributed DBMSs, which was studied in depth during the 1980s and early 1990s. Polyglot persistence approach is similar to the use of mediators in early federated database systems. Recently, some research groups have been working on polyglot persistence platforms. For example, Musketeer [6] provides an intermediate representation between applications and data processing platforms and has the merit of proposing an optimizer for the supported applications and platforms. DBMS+ [7] is another work that aims at embracing several processing and storage platforms for declarative processing. BigDAWG [8] has recently been proposed as a federated system that enables users to run their queries over multiple vertically integrated systems such as column stores, NewSQL engines, and array stores.

The second approach is to develop MMDBMSs to support multiple data models against a single, integrated backend, while meeting the growing requirements for scalability and performance. However, as far as our knowledge, there exist very few research works [12, 20, 21, 28] on the theories and algorithms of MMDBMS. Paper [20] illustrates the query compilation technique for logical and physical design in data management that is relevant to query processing over multiple physical data models in MMDBMS. Paper [21] has introduced the concept of "meta-model" as a framework for defining different data models and specifying translations schema among data models. In this paper, we make the contributions by showing the benefit of leveraging category theory as new theoretical foundation in MMDBMS. We suggest a new mathematical foundation, which not only can capture relational model and relational algebra but also be able to capture many other data models and their algebra, so that the meta-connections among data models and their algebra are transparent from their data applications.

6 Conclusion and Future Work

In this paper, we have discussed the challenges supporting the "Variety" of data; we are advocating an MMDBMS technology with category theory as mathematical foundation. We envision leveraging category theory for multi-model query transformation, viewing processing, in-memory processing and adaptive schema. We also propose a set of shared data infrastructure services in MMDBMS to support "Just-In-Time" multi-model data access autonomously.

Exciting follow-up research can be centered on an in-depth research of category theory into MMDBMS. Of foremost importance is to chart the natural transformation among multiple models to enable transparent cross-model query processing and rewriting. Another of our efforts is aimed at the potential impact of the interplay between category theory and machine learning algorithms on an autonomous data model selection and accesses.

Acknowledgement. Jiaheng Lu is partially supported by the Academy of Finland (No. 310321). Corresponding author and Email: Jiaheng.lu@helsinki.fi.

References

1. W. Spoth, B. S. Arab, E. S. Chan, D. Gawlick, A. Ghoneimy, B. Glavic, B. C. Hammerschmidt, O. Kennedy, S. Lee, Z. H. Liu, X. Niu, Y. Yang. "Adaptive schema databases." In CIDR 2017
2. Z.H. Liu, and D. Gawlick. "Management of Flexible Schema Data in RDBMSs-Opportunities and Limitations for NoSQL-." CIDR. 2015.
3. Lahiri, Tirthankar, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase et al. "Oracle database in-memory: A dual format in-memory database." In Data Engineering (ICDE), 2015
4. Gawlick, Dieter, Eric S. Chan, Adel Ghoneimy, and Zhen Hua Liu. "Mastering situation awareness: The next big challenge?." ACM SIGMOD Record 44, no. 3 (2015): 19-24.
5. David I. Spivak: Database queries and constraints via lifting problems. Mathematical Structures in Computer Science 24(6) (2014)
6. Matthew P. Grosvenor, Allen Clement, and Steven Hand. Musketeer: all for one, one for all in data processing systems. In EuroSys, pages 1-16. 2015
7. Harold Lim, Yuzhang Han, and Shivnath Babu. How to fit when no one size fits. In CIDR. 2013
8. Elmore, Aaron, Jennie Duggan, Mike Stonebraker, Magdalena Balazinska, Ugur Cetintemel, Vijay Gadepally, Jeffrey Heer et al. "A demonstration of the BigDAWG polystore system." Proceedings of the VLDB Endowment 8, no. 12 (2015): 1908-1911
9. Patrick Schultz et al.: Algebraic Databases. CoRR abs/1602.03501 (2016)
10. M. Fleming, R. Gunther, R. Rosebrugh. A database of categories. Journal of Symbolic Computing, 35, 2002
11. R. Wisnesky, D. Spivak. A Functorial Query Language; Presented at Boston Haskell, 2014 http://categoricaldata.net/fql/haskell.pdf
12. S. Abiteboul, et al:Research Directions for Principles of Data Management (Abridged). SIGMOD Record 45(4): 5-17 (2016)
13. Z. H. Liu, et al: Towards a physical XML independent XQuery/SQL/XML engine. PVLDB 1(2): 1356-1367 (2008)
14. Barr Michael, Wells Charles: Category Theory for Computing Science, Reprints in Theory and Applications of Categories, 22 (2012)
15. Da Yan, et al: Big Graph Analytics Platforms. Foundations and Trends in Databases 7(1-2): 1-195 (2017)
16. Jiaheng Lu, Irena Holubová: Multi-model Data Management: What's New and What's Next? EDBT 2017: 602-605
17. World Wide Web Consortium (W3C) https://www.w3.org/
18. Ryan Wisnesky, David I. Spivak, Patrick Schultz, Eswaran Subrahmanian: Functorial Data Migration: From Theory to Practice. CoRR abs/1502.05947 (2015)
19. Riehl, Emily. Category theory in context. Courier Dover Publications, 2017.
20. David Toman, Grant E. Weddell: Fundamentals of Physical Design and Query Compilation. Synthesis Lectures on Data Management, Morgan & Claypool Publishers 2011.
21. Paolo Atzeni, Riccardo Torlone: A metamodel approach for the management of multiple models and translation of schemes. Inf. Syst. 18(6): 349-362 (1993)
22. Property Graph Query Language 1.1 Specification: http://pgql-lang.org/spec/1.1/
23. Z. H. Liu, B. C. Hammerschmidt, D. McMahon, Y. Liu, H. J. Chang: Closing the functional and Performance Gap between SQL and NoSQL. SIGMOD Conference 2016: 227-238

24. J-Y. Girard: Locus Solum: From the rules of logic to logic of rules. In Mathematical Structures in Computer Science, volume11, issue 3, 2001
25. J-Y. Girard: From Foundations to Ludics. In Bulletin of Symbolic Logic, Volume 2, Issue 2, 2003
26. A. Lecomte: Meaning, Logic and Ludics. Imperial College Press, London, 2011
27. Y. Liu, B. Zheng, X.He, Z. Wei, X. Xiao, K. Zheng, J. Lu, "ProbeSim: Scalable Single-Source and Top-k SimRank Computations on Dynamic Graphs". PVLDB 11(1): 14-26 (2017)
28. Jiaheng Lu: Towards Benchmarking Multi-Model Databases. CIDR 2017
29. J. Chen et al: "Big data challenge: a data management perspective". Frontiers Comput. Sci. 7(2): 157-164 (2013)