UDBMS: Road to Unification for Multi-model Data Management

Jiaheng Lu[1(✉)], Zhen Hua Liu[2], Pengfei Xu[1], and Chao Zhang[1]

[1] University of Helsinki, Helsinki, Finland
jiaheng.lu@helsinki.fi
[2] Oracle, Redwood Shore, CA, USA

Abstract. One of the greatest challenges in big data management is the "*Variety*" of the data. The data may be presented in various types and formats: structured, semi-structured and unstructured. For instance, data can be modeled as relational, key-value, and graph models. Having a single data platform for managing both well-structured data and NoSQL data is beneficial to users; this approach reduces significantly integration, migration, development, maintenance, and operational issues. Therefore, a challenging research work is how to develop an efficient consolidated single data management platform covering both NoSQL and relational data to reduce integration issues, simplify operations, and eliminate migration issues. In this paper, we envision novel principles and technologies to handle multiple models of data in one unified database system, including model-agnostic storage, unified query processing and indexes, in-memory structures and multi-model transactions. We discuss our visions as well as present research challenges that we need to address.

1 Introduction

As data in all forms and sizes are critical to making the best possible decisions in businesses, we see the continued growth of demands to manage and analyze massive volume of different types of data. The data may be presented in various types and formats: structured, semi-structured and unstructured [14,16,22]. In the case of structured data, data might be structured as relational, key-value, and graph models [15]. In the case of semi-structured data, data might be represented as XML and JSON documents [19]. Consequently, beyond the traditional relational database (RDBMS), there has been a blooming of different big data management solutions, specialized for different kinds of data and tasks, to name a few, distributed file systems (e.g. Ceph and HDFS), NoSQL data stores (e.g. Bigtable, Redis, Cassandra, Mongodb, Neo4j), and distributed data processing frameworks (e.g. MapReduce, Spark). This is also known for "*one size does not fit all*" argument. However, managing heterogeneous data sources across the systems imposes a big challenge for practitioners in that the deployment of multiple systems has led to a wide diversification of data store interfaces and the loss of a common programming paradigm.

C. Woo et al. (Eds.): ER 2018 Workshops, LNCS 11158, pp. 285–294, 2018.
https://doi.org/10.1007/978-3-030-01391-2_33

Let us consider three application scenarios to illustrate the variety of data. First, consider an application called *customer-360-view*, which often requires to aggregate multiple data sources, including graph data from social networks, document data from product orders and customer information in a relation database. Second, in Oil & Gas industry, a single oil company can produce more than 1.5 TB of diverse data per day. Such data may be structured or semi-structured and come from heterogeneous sources, such as sensors, GPS devices, and other measuring instruments. Third, in health-care: North York hospital needs to process 50 diverse datasets, including structured and unstructured data from clinical, operational and financial systems, and data from social media and public health records. *These emerging applications clearly demand the need to manage multiple-model data in complex, modern applications.*

There exist two approaches to address the challenge of multi-model data management: (i) *polyglot persistence* [10,13] and (ii) *multi-model database*. The first solution uses multiple databases to handle different forms of data and integrates them to provide a unified interface, whereas the second supports multiple data models against a single, integrated backend while meeting the growing requirements for fault tolerance, scalability, and performance. We discuss the issues of both solutions below.

The history of polyglot persistence may trace back to the federation of relational engines [11], or distributed DBMSs, which were extensively studied in depth during the 1980s and early 1990s. Polyglot persistence approach is similar to the use of *mediators* in early federated database systems. For example, Musketeer [10] provides an intermediate representation between applications and data processing platforms. DBMS+ [13] aims at embracing several processing and storage platforms for declarative processing, and 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. Overall, the polyglot persistence solution needs to integrate multiple systems to provide a unified interface, which imposes operational complexity and cost, because the integration of multiple independent databases imposes a significant engineering and operational cost. Further, in order to answer a global query, all of the sub-systems need to remain up, which makes the fault tolerance of the application equal to the weakest component in the stack.

The second approach is to develop a new database to support multiple data models against a single, integrated backend, while meeting the growing requirements for scalability and performance. We observe a recent trend among relational and NoSQL databases in moving away from one single model to multi-model databases. For example, OrientDB is extending graph database to support key-value and json models. ArangoDB is moving from document model to support key-value and graph models. AgensGraph simultaneously supports both the property graph model and the relational model based on a PostgreSQL kernel. Compared to the polyglot persistence, a single multi-model database can reduce the cost of integration, migration, development and maintenance on multiple

systems. Furthermore, during query execution, it can leverage a unified query language directly to access the data stores without query decomposition because of its native store nature.

This paper envisions a multi-model database system, called UDBMS (Unified DataBase Management System), that provides several new components and functions to enable a unified and efficient management of multi-model data. The contributions of this paper are summarized as follows. (1) We envision the architecture of UDBMS system to enable the unified data access and management; (2) We show five model-agnostic properties of UDBMS on data storage, query processing, index structure, in-memory structure and transaction; and (3) we illustrate three new research challenges of UDBMS on schema discovery, model evolution and multi-model data sharding.

In the sequel, we introduce UDBMS's architecture, and walk the readers through the main technical elements of our solutions. Finally, we discuss related works, then conclude.

2 Architecture

The overview of UDBMS architecture is illustrated in Fig. 1.

The main component of the system includes two layers: the *core layer* receives queries from the client and returns the unified results; and the *storage layer*

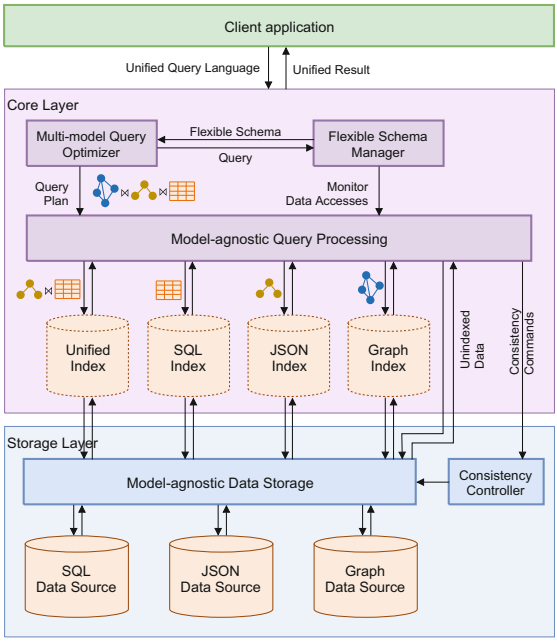

Fig. 1. Architecture of the UDBMS.

loads data with diverse models and stores them in a unified storage platform. We depict the important components in these two layers below:

(i) *Model-agnostic query processing* is responsible for validation and compilation incoming unified queries, as well as query plan generation and optimization. The validation and compilation are supported in cooperation with the flexible schema manager. Moreover, multiple types of *indexes* are developed to efficiently answer ad hoc queries.
(ii) *Flexible schema manager* is to handle the diverse schema of multi-model data. Relational DBMS works according to "*schema-on-write*" principle, which pre-defines the schema prior to feeding data into system. Whereas NoSQL usually agrees "*schema-on-read*", where schema is needed only when reading data from system. To gain a high level of schema flexibility, we continue to use the "schema-on-read" principle by employing a schema manager which tracks data accesses (including read and write) and automatically generates schema and refines them when necessary. By doing this we are able to give an answer without any input schema - we can generate one by ourselves.
(iii) *Model-agnostic data storage* is to provide a model-agnostic abstraction of multi-model data to the accesses of data. The physical data can be stored in different formats and dispersed on a distributed platform, e.g. Hadoop and Spark.
(iv) *Consistency controller* controls the level of data consistency for a single query by using multi-model locks on data. The locks have various types for different models of data, and they support the fine-grained management for better efficiency, e.g., record-, document-, and node/edge-level.

From the above description, it can be seen that UDBMS is different from the existing wrapper-mediator systems [10,13], where data resides in various stores and query execution is divided between the mediator and the wrappers. Instead, UDBMS imports data across the different-data model stores and employs the same abstraction and approach to access and manage them.

3 Under the Hood

UDBMS aims at efficiently managing heterogeneous datasets through a unified set of interface and abstraction. In this section we lay out several model-agnostic properties and discuss how we approach each of them respectively.

3.1 Model-Agnostic Data Storage

Classical RDBMS makes a tight connection between logic data model and physical storage so that the storage engine assumes that data is physically stored in a particular sequence of bytes to support relational access pattern. Though Object-Relational Database Management System (ORDBMS) can be considered as an early version of multi-model DB via handling non-relational data in RDBMS, its starting point is that relational data is the first class citizen. But in

UDBMS, relational data is just one kind of data models, and there is no specific bias towards relational data. Therefore, the storage layer of UDBMS makes no assumption of how multi-model data is internally laid out. It provides a collection of abstraction API that holds a set of objects. Each of the object is accessed by an object id with version based time stamp. There is a key-value interface that supports generic *put()/get()/replace()/delete()* APIs for an object collection. From storage layer perspective, the value is a sequence of bytes which are not interpreted by the storage engine. The sequence of bytes can be a relational row with well-defined schema, or a piece of graph, or an XML document etc.

Specifically, the *get(objectId, timeStamp, objectBytes)* interface returns a sequence of bytes from the storage given an object id. In addition to the object id, the *get()* interface accepts a time stamp snapshot parameter. This parameter identifies the version of the object bytes for that object id that the data storage engine will provide. The *put(objectId, objectBytes)* interface returns a new object id for a sequence of bytes for storage. The *replace(objectId, objectBytes)* interface replaces the content of the object with object id and a new sequence of bytes and returning the new time stamp for the data. The *delete(objectId)* interface deletes the object with the id. However, *get(objectId, oldTimestamp)* may still be able to retrieve the object with the old time stamp. In addition, to support both ACID, BASE or hybrid transactional semantics, the object collection can be declaratively specified as ACID or BASE. The transaction layer then enforces different transaction semantics for the model-agnostic data.

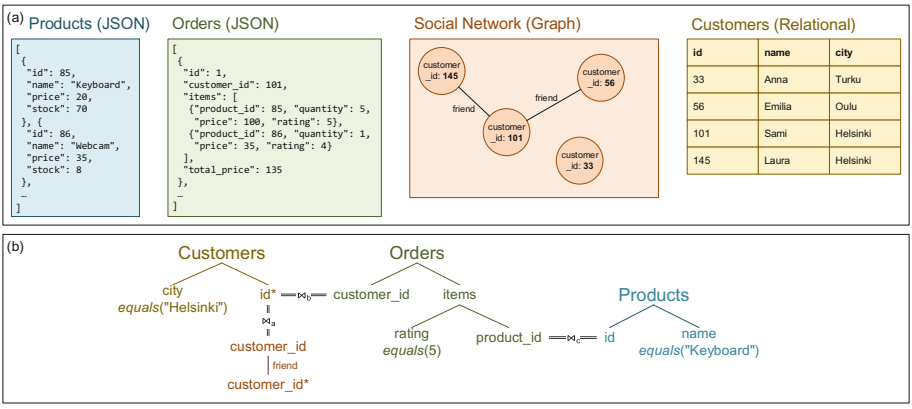

Fig. 2. Example of (a) multi-model datasets and (b) query represented with a forest pattern. The elements marked with asterisk (*) are returned in query results.

3.2 Multi-model Query Processing

The original ORDBMS does not embrace a language to process multi-model data, nor does it address the idea of doing inter-model compilation and optimization. To develop a unified query to accommodate all the data, there are

several existing works towards providing a global language to query multi-model data simultaneously. For instance, SQL++ [21] is proposed to query both JSON native stores and relational data. ArangoDB AQL can be used to retrieve and modify both document and graph data. We use the following example to demonstrate the core structure in a multi-model query.

Example 1. Consider an application involving JSON documents, a relational table and a graph data in Fig. 2(a). One example query is to return *the friends of the customers in Helsinki who bought a keyboard and gave a five-star feedback.* This query can be used for product recommendation. Note that there are three types of joins in the query of Fig. 2(b): *graph-relational* ($\bowtie_a$), *relational-JSON* ($\bowtie_b$) and *JSON-JSON* ($\bowtie_c$) joins. The answer of this query is two pairs of customer IDs: (101,145) and (101,56). □

To process the above query efficiently, as the order of joins can significantly affect the execution time, a query optimizer evaluates available plans and selects the best one. For example, how to decide the join orders among $\bowtie_a$, $\bowtie_b$ and $\bowtie_c$ in Example 1. Therefore, one challenge is to develop new algorithms to select the best query plan for a multi-model query. In addition, statistics, such as *histogram* or *wavelet* can be used to provide detailed information about data distribution for query optimization. The existing statistics techniques on RDBMS are developed based on the static relation schema, but multi-model data requires the diverse and flexible schema. Therefore, it starts to crystallize that dynamic statistics techniques are necessary here to adapt for the frequent schema changes.

3.3 Model-Agnostic Index Structure

Original ORDBMS builds up domain index for each data, cross-domain query join is done by doing separate domain index probes for each domain data, and then joining the index results which are typically at document object ID level. This approach works if we do inter-document object join. However, in UDBMS, such single domain index idea needs to be re-visited if we want to do intra-document object join. For example, to support full text search and relational scalar data search, we need to built up search indexes to incorporate IR-style inverted lists to index various data together. But building a universal search index for all data models requires more deep thoughts. Existing index structures focus upon a single data model, e.g. *B-tree* is used for relational joins, *XB-tree* [4,20] is developed for XML data, and *gIndex* [23] is applicable for graph queries, our visioned system, however, executes queries on more than one data model. Therefore, *how to index multiple data models to accelerate operations such as cross-model filtering and join?* For example, how to support $\bowtie_a$ and $\bowtie_b$ operations in Example 1 for graph-relation and JSON-relation joins efficiently?

In general, we envision two types of auxiliary structures. The first is using inverted index based search index for full-text search. This index integrates domain context aware inverted index for XML and JSON, full text, and leaf scalar relational data together as one unified index. The second is building ad-hoc

global indexes to capture the structural feature in multi-model data to speedup structural query processing.

3.4 Multi-model In-Memory Structure

Classical RDBMS uses buffer cache and assumes that the data layout on disk is the same as what is cached in memory. Nevertheless, UDBMS data storage engine makes no assumption of the domain specific data layout, nor it makes any assumption on how the domain specific data format is when it is cached in memory for fast query. Each domain specific data model may cache the data in different format from what is stored on disk.

Multi-model in-memory cache merely uses callback to delegate to each data-model to get its specific in-memory format. For example, the proposed in-memory data structures for relational database include, to name a few, Oracle columnar layout, IBM DB2 BLU Acceleration and SQL Server In-Memory Columnstore. Through adopting the idea of decoupling the storage format with in-memory query friendly format, UDBMS will provide fast in-memory access for multi-model data using its domain specific query language without worrying about finding the optimal storage model for multi-model data. In fact, there is probably no single best storage format for a domain specific data to satisfy all the workload requirements. For example, columnar layout of the relational data might not always be faster than row layout of the relational data for OLTP query. Another aspect of in-memory cache for domain specific model data is to enable efficient intra-object navigation and traversal when domain specific query language is being evaluated.

3.5 Model-Agnostic Transaction

RDBMS supports ACID guarantee, while NoSQL employs BASE pertaining as the ways for scaling and workaround on CAP theorem. We envision a per-query choice of consistency between ACID and BASE for multi-model data, which is flexible so that the user has a clear understanding and control over the performance as well as the consistency guarantees. To support a hybrid transactional semantics, the model-agnostic object collection can be declaratively specified as ACID or BASE. The transaction layer then enforces different transaction semantics for the data.

Further, to boost the performance of transaction execution, a fine-granular isolation at different levels in multi-model data can achieve the flexibility and performance benefit. For example, objects can be isolated in the forms of subtree locks, subgraph locks, path locks and neighbor node locks. Further, an effective global node labeling scheme can be developed to enable the quick jump to a particular inner data node as required in the lock manager (e.g. to support *getParent* operations in a tree).

3.6 Other Research Challenges

Schema Discovery. Original ORDBMS assumes the perfect schema based world. Semi-structure data and unstructured data challenge ORDBMS with schema-less design. We understand the value of NoSQL point of schema-less DB development, but further enhances it to argue that schema-less for write is half of the story, schema-rich for query is the other half of the story via auto-schema discovery. Therefore, UDBMS is expected to support schema discovery interface. Enhancing schema discovery for multi-model data is a new challenge which involves knowledge and techniques from neighboring research communities such as data mining, machine learning and human-computer interaction.

Model Evolution is a unique challenge in a multi-model database [17]. With the increasing maturity of NoSQL databases, a plethora of applications turn to store data with JSON documents or graph representations. But the legacy data are still stored in the traditional RDBMS. Thus, the model evolution may affect the usability of queries and applications developed on the RDBMS. Therefore, a research challenge is how to perform model mapping and query rewriting to automatically handle such model evolution. Note that model evolution is a more complicated than schema evolution on RDBMS, because it raises the issues involving both the attribute difference and the structural difference.

Multi-model Sharding is a method for distributing data across multiple machines. A relational database shard is a horizontal partition of data and each shard is held on a separate database server instance to spread load. But in the scenario of multi-model data management, do we support inter-object or intra-object sharding ? It is easy to support inter-object sharding. But if a graph or a tree is a big object, then we need to consider about intra-object sharding. Therefore, the distributed data sharding technology needs further investigation in UDBMS.

4 Related Work

Heterogeneous Query Processing. The interests of providing a unified data access and processing interface for heterogeneous data sets have been extensively explored in previous systems and prototypes [6,9,18]. For example, BigIntegrator [24] supports SQL-like queries that combines data in Bigtable stores in the cloud and data in relational stores; Forward presents SQL++ [21], an SQL-like language designed to unify the data model and query language capabilities of NoSQL and relational databases; Dremel [2] uses semi-structure data model, however, it can execute queries based on a flattened columnar storage. Vertexica [12] runs graph queries in a relational database; Asterix follows the motto "*one size fits a bunch*" and has built a data model-agnostic query compiler substrate, called Algebricks [3], and has used it to implement three different query languages: HiveQL, AQL, and XQuery, on different format of data. Compared to the previous works which focus on specific models, our approach is more generic, with several model-agnostics principles (for data storage, query processing and

index structures) that enable the efficient heterogeneous query processing for multiple data stores.

Tightly-Coupled Multistore System. Recent works (e.g. Polybase [7], HadoopDB [1] and Estocada [5]) also demonstrate the performance benefits of using seamless integrated multi-stores, these systems aim at efficient management of structured and unstructured data for big data analytic, particularly for integration of HDFS and RDBMS data. Different from the purpose of combining relational and distributed unstructured stores, UDBMS focus on the unification of relational and transactional NoSQL stores for on-line hybrid OLTP and OLAP data processing.

5 Conclusion and Future Works

> "*The intellect seeking after a unified theory cannot rest content with the assumption that there exist two distinct fields totally independent of each other by their nature.*"
> — Albert Einstein in his Nobel lecture in 1923

In this paper, we present our visions to build a system to query, index and update multi-model data in a unified fashion. While the road to unification is full of challenges, this paper has laid down our visions to build a UDBMS with model agnostic properties and structures. In the future work, we shall investigate the following three categories of challenges on UDBMS: diversity, extensibility and flexibility.

(1) Diversity: The first challenge is the "*diversity*" of multi-model data. The existing results for query optimization and transaction model mainly work on a single model, either structured or semi-structured data. The highly diverse nature of multi-model data makes a unified system complicated and fascinated.

(2) Extensibility: The second challenge is to identify the boundary of UDBMS. In this paper, we envision a unified system for several types of data, i.e. relation, JSON, XML and graph. A further question is how to adopt more types of data such as streaming data and time series data. This calls for the future research on the extensibility of a multi-model system.

(3) Flexibility: Finally, NoSQL DBMS can support schema-less for storage, and schema-rich for query according to the automatically discovered schema. It would be interesting to explore the model-agnostic storage and query processing in UDBMS – in particular, we call this "*what we store is* ***not*** *what we get*".

Acknowledgment. Contact email: Jiaheng.Lu@helsinki.fi. This work is partially supported by Academy of Finland (Project No. 310321).

References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D.J., Rasin, A., Silberschatz, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. PVLDB **2**(1), 922–933 (2009)
2. Afrati, F.N.: Storing and querying tree-structured records in Dremel. PVLDB **7**(12), 1131–1142 (2014)
3. Borkar, V.R., et al.: Algebricks: a data model-agnostic compiler backend for big data languages. In: ACM SoCC, pp. 422–433 (2015)
4. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: ACM SIGMOD, pp. 310–321 (2002)
5. Bugiotti, F., Bursztyn, D., Deutsch, A., Ileana, I., Manolescu, I.: Invisible glue: scalable self-tunning multi-stores. In: CIDR (2015)
6. Chen, J., et al.: Big data challenge: a data management perspective. Front. Comput. Sci. **7**(2), 157–164 (2013)
7. DeWitt, D.J., et al.: Split query processing in polybase. In: SIGMOD, pp. 1255–1266 (2013)
8. Elmore, A.J., et al.: A demonstration of the BigDAWG polystore system. PVLDB **8**(12), 1908–1911 (2015)
9. Franklin, M.J., Halevy, A.Y., Maier, D.: From databases to dataspaces: a new abstraction for information management. SIGMOD Rec. **34**(4), 27–33 (2005)
10. Gog, I., et al.: Musketeer: all for one, one for all in data processing systems. In: EuroSys, pp. 1–16 (2015)
11. Heimbigner, D., McLeod, D.: A federated architecture for information management. ACM Trans. Inf. Syst. **3**(3), 253–278 (1985)
12. Jindal, A., et al.: VERTEXICA: your relational friend for graph analytics!. PVLDB **7**(13), 1669–1672 (2014)
13. Lim, H., Han, Y., Babu, S.: How to fit when no one size fits. In: CIDR (2013)
14. Lin, C., Lu, J., Wei, Z., Wang, J., Xiao, X.: Optimal algorithms for selecting top-k combinations of attributes: theory and applications. VLDB J. **27**(1), 27–52 (2018)
15. Liu, Y., Lu, J., Yang, H., Xiao, X., Wei, Z.: Towards maximum independent sets on massive graphs. PVLDB **8**(13), 2122–2133 (2015)
16. Liu, Y., et al.: ProbeSim: scalable single-source and top-k simrank computations on dynamic graphs. PVLDB **11**(1), 14–26 (2017)
17. Lu, J.: Towards benchmarking multi-model databases. In: CIDR (2017)
18. Lu, J., Holubová, I.: Multi-model data management: what's new and what's next? In: EDBT, pp. 602–605 (2017)
19. Lu, J., Ling, T.W., Bao, Z., Wang, C.: Extended XML tree pattern matching: theories and algorithms. IEEE Trans. Knowl. Data Eng. **23**(3), 402–416 (2011)
20. Lu, J., Ling, T.W., Chan, C.Y., Chen, T.: From region encoding to extended dewey: on efficient processing of XML twig pattern matching. In: VLDB, pp. 193–204 (2005)
21. Ong, K.W., Papakonstantinou, Y., Vernoux, R.: The SQL++ semi-structured data model and query language: A capabilities survey of SQL-on-Hadoop, NoSQL and NewSQL databases. CoRR abs/1405.3631 (2014)
22. Xu, P., Lu, J.: Top-k string auto-completion with synonyms. In: Candan, S., Chen, L., Pedersen, T.B., Chang, L., Hua, W. (eds.) DASFAA 2017. LNCS, vol. 10178, pp. 202–218. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55699-4_13
23. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In: SIGMOD, pp. 335–346 (2004)
24. Zhu, M., Risch, T.: Querying combined cloud-based and relational databases. In: CSC, pp. 330–335 (2011)