UniBench: A Benchmark for Multi-model Database Management Systems

Chao Zhang, Jiaheng Lu[(✉)], Pengfei Xu, and Yuxing Chen

Department of Computer Science, University of Helsinki, Helsinki, Finland
Jiaheng.Lu@helsinki.fi

Abstract. Unlike traditional database management systems which are organized around a single data model, a multi-model database (MMDB) utilizes a single, integrated back-end to support multiple data models, such as document, graph, relational, and key-value. As more and more platforms are proposed to deal with multi-model data, it becomes crucial to establish a benchmark for evaluating the performance and usability of MMDBs. Previous benchmarks, however, are inadequate for such scenario because they lack a comprehensive consideration for multiple models of data. In this paper, we present a benchmark, called UniBench, with the goal of facilitating a holistic and rigorous evaluation of MMDBs. UniBench consists of a mixed data model, a synthetic multi-model data generator, and a set of core workloads. Specifically, the data model simulates an emerging application: Social Commerce, a Web-based application combining E-commerce and social media. The data generator provides diverse data format including JSON, XML, key-value, tabular, and graph. The workloads are comprised of a set of multi-model queries and transactions, aiming to cover essential aspects of multi-model data management. We implemented all workloads on ArangoDB and OrientDB to illustrate the feasibility of our proposed benchmarking system and show the learned lessons through the evaluation of these two multi-model databases. The source code and data of this benchmark can be downloaded at http://udbms.cs.helsinki.fi/bench/.

1 Introduction

Multi-Model DataBase (MMDB) is an emerging trend for the database management system [16,17], which utilizes a single platform to manage data stored in different models, such as document, graph, relational, and key-value. Compared to the polyglot persistence technology [24] that employs separate data stores to satisfy various use cases, MMDB is considered as the next generation of data management system incorporating flexibility, scalability, and consistency. The recent Gartner Magic quadrant [9] for operational database management systems predicts that, in the near future, all leading operational DBMSs will offer multiple data models in a unified platform. MMDB is beneficial for modern applications that require dealing with heterogeneous data sources while embracing the

agile development. For instance, in a *Social Commerce* application [27], enterprises often gain business insights by integrating graphs from social networks, documents from the purchase history, and tables from customer information. Data scientists usually write scripts for each data model separately, then wrangles them into a unified form to proceed with real-time and OLAP analysis. However, as the scale and complexity of data increase, such method becomes tedious and inefficient. By leveraging the power of MMDB, one can easily ingest and analyze heterogeneous data in real time and hence swiftly adjust the operational strategy.

Database benchmark becomes an essential tool for the evaluation and comparison of DBMSs since the advent of Wisconsin benchmark [5] in the early 1980s. Since then, many database benchmarks have been proposed by academia and industry for various evaluation goals, such as TPC-C [25] for RDBMSs, TPC-DI [21] for data integration; OO7 benchmark [2] for object-oriented DBMSs, and XML benchmark systems [15,23] for XML DBMSs. More recently, the NoSQL and big data movement in the late 2000s brought the arrival of the next generation of benchmarks, such as YCSB benchmark [4] for cloud serving systems, LDBC [6] for Graph and RDF DBMSs, BigBench [3,10] for big data systems. However, those general-purpose or micro benchmarks are not designed for MMDBs. As more and more platforms are proposed to deal with multi-model data, it becomes important to have a benchmark for evaluating the performance of MMDBs and comparing different multi-model approaches.

In general, there are two challenges evaluating the performance of MMDBs:

Fig. 1. Unibench data model

The first challenge is to generate synthetic multi-model data. First, existing data generators cannot be directly adopted to evaluate MMDBs because they only involve one model. Besides, combining them reasonably is a difficult task since each generator simulates a particular scenario. In this study, we develop a new data generator to provide correlated data 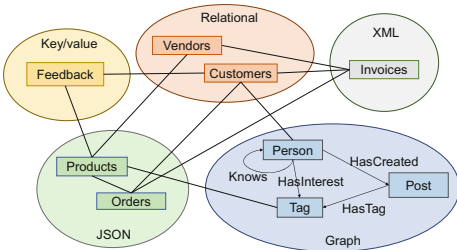in diverse models. As shown in Fig. 1, our benchmark system consists of five data models, i.e., Graph, Relational, JSON, Key-value, and XML. It simulates a *social commerce* scenario [27] that combines the social network with the E-commerce context. The relational model includes the structured *customers* and *vendors*, JSON model contains the semi-structured *orders* and *products*. The social network is modeled as graph, which includes three entities and four relations. i.e., *person, post, tag, person_knows_person, person_has_tag, person_create_post, post_has_tag*. *Feedback* and *Invoices* are modeled as key-value and XML, respectively. These also have correlations across the data models. For instance, customer *knows* friends (relational correlates with the graph model), customer *makes* transactions. (JSON

correlates with relational model). Furthermore, we propose a three-phase framework to simulate customers' behaviors in social commerce. This framework consists of purchase, propagation-purchase, and re-purchase, which takes into account a variety of factors to generate the Power-law distribution data that are widely seen in real life. Particularly, we propose a new probabilistic model CLVSC (Customer Lifetime Value in Social Commerce) to make fine-grained predictions in the third phase.

The second challenge is to design multi-model workloads. Such workloads are the fundamental operations in many complex and modern applications. However, little attention has been paid to study them. It is non-trivial to design the workloads which not only cover the most important paradigms of multi-model query processing but also simulate realistic use cases. In this regard, we first simulate meaningful business cases in social commerce by dividing them into four layers: *individual*, *conversation*, *community*, and *commerce*. Then we define a set of multi-model queries and transactions based on the choke point technique [22], which tests the weak points of databases to make the benchmark challenging and interesting. Choke points of our benchmark workloads involve performances of the multi-model aggregation, join, and transaction, demanding the database to determine the optimal multi-model join order, handle the complex aggregations, and guarantee the concurrency and efficiency simultaneously.

We summarize our contributions as follows:

1. We develop a new data generator, which provides correlated data in diverse data models. We also propose a three-phase framework to generate data for modeling the customers' behaviors in social commerce. We implement the generator on the top of Spark and Hadoop to provide efficiency and scalability.
2. We design a set of multi-model workloads including ten queries and two transactions from technical and business perspectives.
3. We implement proposed workloads and conduct experiments on two MMDBs: ArangoDB [1] and OrientDB [19]. We analytically report the performance comparison and our learned lessons.

The rest of this paper is divided as follows. Section 2 introduces the background and related work. Section 3 illustrates the workflow of data generation. Section 4 presents the multi-model workload in detail. The experimental results are shown in Section 5. Finally, Sect. 6 concludes this work.

2   Background and Related Work

Background. Multi-model data management is proposed to address the "*Variety*" challenge of data in a complex world. The first evolution is the prevalence of *Polyglot Persistence* [24] method, which exploits numerous databases to handle different forms of data and integrates them to provide a unified interface. Unfortunately, such method imposes further operational complexity and cost, because the need for integrating multiple databases has a significant engineering and operational overhead. The drawback of *Polyglot Persistence* leads to the

Table 1. Comparison of multi-model DBMSs

System	Query language	Primary model	Secondary model	Storage strategy
AgensGraph	OpenCypher, SQL	Relational	Graph, JSON	One engine
ArangoDB	AQL	JSON	Graph, Key-value	One engine
OrientDB	SQL-like	Graph	JSON, Key-value	One engine
Marklogic	Xpath	XML	JSON, RDF	One engine
Redis	API	Key-value	Graph, JSON	One engine
NitrosBase	SparQL, SQL	RDF	Graph, JSON, Key-value	One engine
Datastax	CQL	Column	JSON, Graph	Multiple engines
DynamoDB	API, SQL	-	JSON, Graph, Key-value	Multiple engines
CosmosDB	API, SQL	-	**ALL** but XML	Multiple engines
Oracle 12c	SQL-extension	Relational	**ALL**	**Both**

second evolution of multi-model data management. First, many SQL-extension ecosystems and NoSQL systems have been transformed to multi-model systems by integrating additional engines or functions into a unified platform for supporting additional models. On the other hand, there emerge many native multi-model databases, e.g., ArangoDB, AgensGraph, OrientDB. These systems utilize a single store to manage the multi-model data, along with a unified or hybrid query language. Table 1 shows the representatives of MMDBs compared by several properties, namely, query language, primary model, secondary model, and storage strategy. The secondary model of each system is extended in the second evolution. Redis, for example, adds JSON and graph to its key-value store. On the other hand, DynamoDB employs several engines to support multiple models including JSON, graph, and key-value, and it has no specified primary model because each model is regarded as the first-class citizen.

Related Work. There are a few works on multi-model data modeling, data generation and database benchmarking. In [16], we have envisioned a multi-model database benchmark system (but without any detailed solution and implementation). Regarding the data modeling and data generation, TPC-DI [21] features a multi-model input data including XML, CSV, and textual parts, which is used to evaluate the transformation cost in data integration process. Also, Bigbench [10] incorporates semi-structured (logs) and unstructured data (reviews) into TPC-DS's structured data. However, no consideration was given to JSON and graph, which are currently two most popular models in data management. As for the performance evaluation, several evaluation efforts [18,20] have been done on multi-model databases recently. Nevertheless, they only focus on simple workloads, such as CRUD operation, aggregation, graph depth traversal, which are inadequate since they do not account for complex workloads concerning multi-model characteristics.

The most relevant work about our approach is LDBC social network benchmark [6]. First, our graph generation is based on LDBC [6], we choose it as

the starting point for its scalability and rich semantics in simulating social networks. It also supports the generation of correlated graph by leveraging the MapReduce paradigm of Hadoop. However, since our goal is concentrating on benchmarking multi-model databases rather than graph databases, we have simplified the graph complexity to better fit our goal. Moreover, in order to generate the E-commerce transactions with associated graph entities, we replace some of its dictionaries by collecting commerce metadata from Amazon review [14] and DBpedia dataset [13]. Second, our workload design is motivated by LDBC [6], TPC-C [25] and Bigbench [10]. In particular, LDBC follows the graph-based choke points approach, and Bigbench focuses on the business questions in five main categories. Motivated by these two design principles, we also propose the choke-point and business-driven query design. For the transaction design, despite the business cases of two proposed transactions e.g., New Order and Payment are similar to those in TPC-C [25], the data involved in our benchmark systems come from multiple data models. Therefore we focus on the multi-model transactions rather than single-model transactions.

3 Data Generation

In this section, we introduce the process of multi-model data generation. Figure 2 shows our three-phase data generation framework. Specifically, (i) in *Purchase* phase, LDBC [6] obtains metadata from the repository, then generates graph data and initial interests of persons. These data is feed to our generator to produce transaction data. (ii) In *Propagation-Purchase* phase, interests of cold-start customers are generated based on information obtained in the previous phase. (iii) In *Re-purchase* phase, interests of all customers will be generated based on CLVSV model, which is discussed shortly. In each phase, we generate transaction data according to the interests of customers and unite all three portions as an integral part of the multi-model dataset. The entire generation process is presented in Algorithm 1, and discussed in detail as follows:

Fig. 2. Data generation workflow.

Algorithm 1. Data Generation

Input: Scale factor f, constant c controlling number of transaction, real value
λ as Poisson parameter, and meta data
Output: Multi-model dataset D

1 $G \leftarrow LDBC(f)$ `// graph data generated by the LDBC generator`
2 $R \leftarrow$ relational data transformed from graph and meta data
3 $L_1 \leftarrow$ initial list of purchase interest of persons from G
4 $J, X, KV, D \leftarrow \varnothing$ `// initial sets of JSON, XML, Key-value, and output`
5 $P_t, P_f \leftarrow$ persons having interests, persons having no interests
6 **foreach** $p \in P_t$ **do**
7 $count \leftarrow L_1/c$
8   **while** $count \neq 0$ **do**
9 $r \leftarrow Poisson(\lambda)$
10 $count \leftarrow count - 1$
11 $J, X, KV \leftarrow Purchase(r, p)$ `// generate transaction data`
12 $D \leftarrow D \cup J \cup X \cup KV$
13 BayesModel $\leftarrow$ fit the bayes model based on R and KV
14 **foreach** $p \in P_f$ **do**
15 $L_2 \leftarrow$ generate new interest list based on equation 1
16 $D \leftarrow D \cup Propagation\text{-}Purchase(p, L_2)$
17 CLVSC $\leftarrow$ fit the CLVSC model based on a small portion of samples
18 **foreach** $p \in P_t \cup P_f$ **do**
19 $L_3 \leftarrow \mathrm{CLVSC}(J, G)$ `// generate new interests by CLVSC model`
20 $D \leftarrow D \cup Re\text{-}Purchase(p, L_3)$ `// generate new transaction data`
21 **return** D

3.1   Purchase

In this phase, we consider two factors when generating the data. First, persons
usually buy products based on their interests. Second, persons with more inter-
ests are more likely to buy products than others. The person's interests for the
products are generated by the LDBC. This phase is implemented on the top of
Spark SQL using Scala, which utilizes a plentiful APIs and UDFs to generate
the multi-model data. Specifically, we first determine the number of transactions
for each person by dividing the number of their interests with a constant c, then
select the size for each transaction from a Poisson distribution with parameter λ,
finally assign items to each transaction by randomly choosing items from their
interest sets. The *orders* will be output in JSON format with an embedded item
array of orderline. Meanwhile, The *invoices* will be generated with the same
information but in XML format. In addition, we randomly select the product's
real review and corresponding rating from the Amazon dataset as the feedback.
Consequently, our data consist of five models: *social network* (Graph), *vendor*
and *customer* (Relation), *order and product* (JSON), *invoice* (XML), *feedback*
(Key-value).

3.2   Propagation-Purchase

In this phase, we incorporate two ingredients from previous data generation: (i) person's basic demographic data, e.g., gender, age, location. (ii) feedback of friends. This is motivated by the observation that people with same attributes more likely have the same behaviors, and people also trust the product recommendations from friends. The scoring function is defined as follow:

$$S_{ui} = \sum_k k \times Pr(R_{ui} = k | A = a_u) + E(R_{vi} : \forall v \in N(u)) \tag{1}$$

where $\sum_k k \times Pr(R_{ui} = k | A = a_u)$ is the expectation of the probability distribution of the target user u's rating on the target item i, and $A = \{a_1, a_2, \ldots, a_m\}$ is user attribute set computed based on Naive Bayesian method. The latter part $E(R_{vi} : \forall v \in N(u))$ is the expectation of u's friends' rating distribution on the target item, where $N(u)$ is the friends set of user u, and the item i is from the purchase transaction of friends. To train the bayes model, we implemented our techniques using Python's scikit-learn, which takes users' profiles and rating history from the previous phase as the training set. For each person without interests, we take the items rated by their friends as the candidate set, then rank them using Eq. (1). Finally, we take the first n percent portion as the new interests, and then generate the new transactions the same as the process in the purchase phase.

3.3   Re-purchase

The CLV (Customer Lifetime Value) model [11] is proposed to address the RFM's limitation in forecasting non-contractual customer behavior. We propose a new probabilistic model CLVSC (Customer Lifetime Value in Social Commerce) to make fine-grained predictions by incorporating the customer's social activities regarding the brand. In general, the CLVSC is comprised of three components: the expected number of behaviors, the expected monetary value, and the expected positive social engagement of customer. The scoring function for CLVSC is defined as follow:

$$\begin{aligned} S_{ib}(CLVSC) = & E(X^* \,|\, n^*, x', n, m, \alpha, \beta, \gamma, \delta) \\ & \times \left(E(M \,|\, p, q, \upsilon, m_x, x) + E(S \,|\, \bar{s}, \theta, \tau) \right) \end{aligned} \tag{2}$$

where i and b are the customer and brand index, respectively,

$E(X^* \,|\, n^*, x', n, m, \alpha, \beta, \gamma, \delta)$ denote the expected number of behaviors over the next n^* periods by a customer with observed behavior history (x', n, m), where x' is the number of behavior that occurred in n period, with the last behavior m $\leqslant$ n; (α, β) and (γ, δ) are the beta distribution parameters for active probability and inactive probability respectively, the behavior is either the purchase or the post. Utilizing the beta-geometric/beta-binomial (BG/BB) [7] model, we have

$$
\begin{aligned}
& E(X^* \mid n^*, x', n, m, \alpha, \beta, \gamma, \delta) \\
& = \frac{B(\alpha + x + 1, \beta + n - x)}{B(\alpha, \beta)} \\
& \times \frac{B(\gamma - 1, \delta + n + 1) - B(\gamma - 1, \delta + n + n^* + 1)}{B(\gamma, \delta)} \\
& \div L(\alpha, \beta, \gamma, \delta | x, n, m)
\end{aligned}
\tag{3}
$$

where $L(\cdot)$ is the likelihood function. This result is derived from taking the expectation over the joint posterior distribution of active probability and inactive probability.

$E(M \mid p, q, \upsilon, m_x, x)$ denote the expected monetary value. Following the Fader, Hardie, and Berger's approach [8] of adding monetary value, we have

$$
\begin{aligned}
& E(M \mid p, q, \upsilon, m_x, x) \\
& = \left(\frac{q - 1}{px + q - 1} \right) \frac{\upsilon p}{q - 1} + \left(\frac{px}{px + q - 1} \right) m_x
\end{aligned}
\tag{4}
$$

$E(S \mid \bar{s}, \theta, \tau)$ denote the expected social engagement of customer, we assume that the number of social engagement of customer follows a Poisson process with rate λ, and heterogeneity in λ follows a gamma distribution with shape parameter θ and rate parameter τ across customers. According to the conjugation of Poisson-gamma model, the point estimate $E(S \mid \bar{s}, \theta, \tau)$ can be factorized as follow,

$$
E(S \mid \bar{s}, \theta, \tau) = \theta' \tau' = \frac{\tau}{1 + \tau} \bar{s} + \frac{\tau}{1 + \tau} \theta \tau
\tag{5}
$$

The resulting point estimate is therefore a weighted average of the sample mean $\bar{s}$ and the prior mean $\theta \tau$.

We implemented the CLVSC model using R's BTYD package [26], which takes a small portion of samples from the previous phases as the training set. For all persons, we estimate their interests of brands, then acquire the m interests from top n brands, finally generate the new transactions the same as the process in the purchase phase.

4 Workload

The UniBench workload consists of a set of complex read-only queries and read-write transactions that involve at least two data models, aiming to cover different business cases and technical perspectives. More specifically, as for business cases, they fall into four main levers [12]: *individual*, *conversation*, *community*, and *commerce*. In these four levers, common-used business cases in different granularity are rendered. Regarding technical perspectives, they are designed based on the *choke-point* technique [22] which combines common technical challenges with new intractable problems for the multi-model query processing, ranging from the conjunctive queries (OLTP) to analysis (OLAP) workloads. Their characteristics are summarized in Table 2. Note that in the description column, the italic and bold texts denote the intended input and output data, respectively.

Table 2. Characteristics of workload

Label	Business category	Technique dimension	Description
Q1	Individual	Perform point query on a customer's all multi-model data	For a given *customer*, find her **profile**, **orders**, **feedback**, and **posts**
Q2	Conversation	Join data from Relation, Graph, and JSON	For a given *product*, find the **persons** who had bought it and posted on it
Q3	Conversation	Join data from Relation, Graph, and Key-value, filter structured and unstructured data	For a given *product*, find **persons** who have commented and posted on it, and detect negative **sentiments** from them
Q4	Community	Aggregate and sort the JSON order, Perform the 3-hop graph traversal in the subgraph, return the intersection of two sets	Find the top-2 **persons** who spend the highest amount of money in orders. Then for each person, traverse her knows-graph with 3-hop to find the friends, and finally return the common **friends** of these two persons
Q5	Community	Join data from Relation, Graph, and Key-value with two predicates, recursive path query for Graph, embedded array operation for JSON, and composited-key lookup for Key-value	Given a start *customer* and a product *category*, find **persons** who are this customer's friends within 3-hop friendships in knows-graph, and they have bought products in the given category. Finally, return **feedback** with the 5-rating review of those bought products
Q6	Community	Perform the shortest path calculations between two nodes, find the correlated JSON orders of nodes in the path, aggregation on returned JSON orders	Given *customer 1* and *customer 2*, find persons in the **shortest path** between them in the subgraph, and return the TOP 3 **best sellers** from all these persons' purchases
Q7	Commerce	Join data from Relation, JSON and Key-value, compare the aggregation results between two periods, identify the reviews with negative sentiment	For the *products* of a given *vendor* with declining sales compare to the former quarter, analyze the **reviews** for these items to see if there are any negative sentiments

Table 2. (*continued*)

Label	Business category	Technique dimension	Description
Q8	Commerce	Perform the embedded array filtering and aggregation on JSON order, aggregate the correlated graph data for each records	For all the *products* of a given *category* during a given year, compute its **total sales amount**, and measure its **popularity** in the social media
Q9	Commerce	Perform the embedded array filtering, aggregation, and sorting on JSON order, then find the correlated graph data	Find top-3 **companies** who have the largest amount of sales at one *country*, for each company, compare the number of the male and female customers, and return the most recent **posts** of them
Q10	Commerce	Perform the aggregation and sort on graph data, then find the correlated Key-value and JSON data	Find the top-10 most active **persons** by aggregating the *posts* during the last year, then calculate their **RFM (Recency, Frequency, Monetary) value** in the same period, and return their recent **reviews** and tags of **interest**
T1	New order transaction	Check the ACID properties and evaluate the efficiency on read-heavy multi-model transaction that involves JSON and XML	(i) Create and insert the **order**, (ii) update the quantity of involved **products**, (iii) insert the **invoice**
T2	Payment transaction	Check the ACID properties and evaluate the efficiency on write-heavy multi-model transaction that involves Relation, JSON and XML	(i) Retrieve the unpaid **order**, (ii) update the balance of the **seller** and **buyer**, (iii) update the **order** status to paid, (iv) update the related **invoice**

4.1   Business Cases

We identify two transactions and four layers of queries that include ten multi-model queries to simulate realistic business cases in social commerce. Specifically, the two transactions, namely, *New Order* and *Payment* transactions, simulates the huge parallel transactions for online shopping. They represent heavy-weight, read-write transactions with a high frequency of execution to satisfy on-line users. As for multi-model queries, the **individual level** mimics the case that

companies build a 360-degree customer view by gathering data from customer's multiple sources. There is one query for this level. **conversation level** focus on analyzing the customer's semi-structured and unstructured data, including Query 2 and 3. The two queries are commonly used for the company to capture customer's sentiment polarity from the feedback and then adjust the online advertising or operation strategy. Query 4, 5, 6, in the **community level** target at two areas: mining common purchase patterns in a community and analyzing the community's influence on the individual's purchase behaviors. Finally, **commerce level** aims at the assortment optimization and performance transparency. Specifically, Query 7, 8, 9 identify products or vendors with downward or upward performance and then find the cause for improvements. Query 10 is to compute the Recency, Frequency, Monetary (RFM) value of customers regarding the vendor, and then find the common tags in the posts.

4.2   Technical Dimensions

Our workload design is based on the *choke point* technique that tests many aspects of the database when handling the query. Typically, these aspects may concern different components of databases, such as the query optimizer, the execution engine, and the storage system. Moreover, the choke points in our workload not only involve common query processing challenges for the traditional database systems but also take a few new problems of multi-model query processing. Here we list three key points:

Choosing the Right Join Type and Order. Determining the proper join type and order for multi-model queries is a new and non-trivial problem. This is because it demands the query optimizer to estimate the cardinality with respect to involved models. Moreover, it needs the query optimizer to judiciously determine the optimal join order for multi-model query. The execution time of different join orders and types may vary by orders of magnitude due to the domination of different data model. Therefore, this choke point tests the query optimizer's ability to find an optimal join type and order for the multi-model query. In our proposed workload, all the queries involve multiple joins across different data model.

Performing Complex Aggregation. This choke-point includes two types of queries concerning the complex aggregation. The first type is the aggregation towards the complex data structure which requires MMDB to deal with schema-agnostic data when proceeding with aggregation. The second one is the query with subsequent aggregations, where the results of an aggregation serve as the input of another aggregation. Also, these aggregations involve the union of multiple models' results. For instance, Query 10 requires the MMDB to access the product array in the JSON orders when processing the first aggregation. Then the results will be an input for the second aggregation in the Graph.

Ensuring the Consistency and Efficiency. A database transaction should possess ACID properties. Therefore, this choke-point tests the ability of the execution engine and the storage system to find an appropriate concurrency control technique to guarantee the consistency and efficiency. In particular, the transactions not only involve read-write operations on multiple entities but also require the MMDB to guarantee the consistency across the data model.

4.3   Example

To illustrate our choke-point-based design of queries, we take Query 5 (in Fig. 3) as an example to explain the technical challenge under the hood. Query 5 is that: *Given a start customer and a product category, find persons who are this customer's friends within 3-hop friendships in Knows graph, besides, they have bought products in the given category. Finally, return the feedback with the 5-rating review of those bought products.*
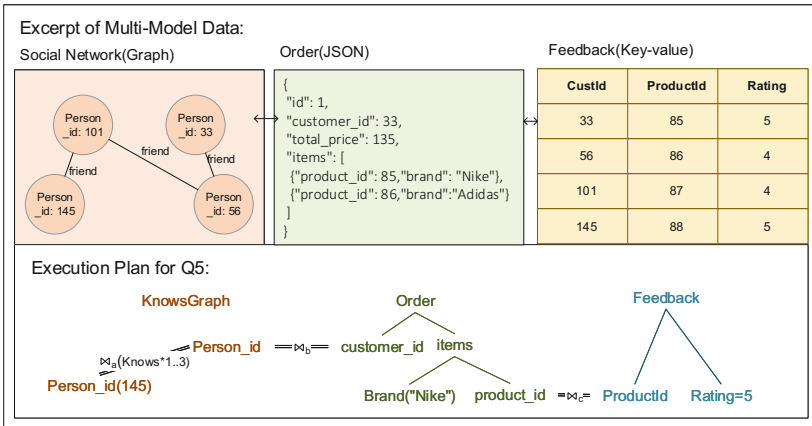
Fig. 3. Example of multi-model join

As Fig. 3 depicts, this query involves three data models: customer with 3-hop friends (*Graph*), order embedded with an item list (*JSON*), and customer's feedback (*Key-value*). From the business perspective, it can be used to explain the recommendation model for better transparency and user experience. From the technical dimension, there are three types of joins in the query: Graph-Graph ($\bowtie_a$), Graph-JSON ($\bowtie_b$) and JSON-KV ($\bowtie_c$) join. Nevertheless, as the order of filters and joins can affect the execution time, an important task for the query optimizer is to evaluate available plans and select the best one. Note that picking a wrong join order makes the performance drastically worse. For example, when there is no qualified tuple in the orders, traversing one thousand tuples in the graph and looking up thousands of key-value pairs would be a bad choice. A judicious way for this case is to filter the orders with given parameters, and

avoid the graph traversal and index lookup for key-value pairs when there are no valid orders. Furthermore, Query 5 is challenging also because each model arises a cardinality estimation issue to the query optimizer, i.e., recursive path query for Graph, embedded array operation for JSON, and composite-key lookup for key-value.

5   Experiments

In this section, we report our experimental results, including the performance of data generation and the benchmark results. In the case of the setup, we generate the synthetic data on a cluster of three machines, each with double 4-core Xeon-E5540 CPU, 32 GB RAM, and 500 GB HDD. In addition, we conduct all benchmark experiments on another machine with double 6-core Xeon-E5649 CPU, 100 GB RAM, and 500 GB HDD. The client machine has a 4-core i5-4590 CPU with 16 GB RAM. We select two representative MMDBs: OrientDB and ArangoDB with community version 2.2.16 and 3.3.7. On the client-side, we develop a Node.js program integrated with each DB's official driver. All benchmark workloads are implemented in the program (except for the OrientDB transaction, which can only be fully supported using JAVA API at present).

5.1   Data Generation

Table 3 presents characteristics of three generated datasets, each of which consists of five data models. Unibench defines a set of scale factors (SFs), targeting systems of different sizes. The size of the resulting dataset is mainly affected by the number of persons (Relational entries). For benchmarking the databases, we leverage the data generator to produce three datasets with roughly size 1 GB, 10 GB, and 30 GB by using scale factors 1, 10, and 30, respectively. In the case of efficiency, experiment results suggest the data generator produced 1 GB and 10 GB multi-model datasets in 10 and 40 min, on our 8-core machine running MapReduce and Spark in "pseudo-distributed" mode. In terms of scalability, we successfully generate 30G multi-model data within 60 min on our three-node cluster.

Table 3. Characteristics of datasets.

SF	Generation time (min)	Number ($\times 10^4$) & size in megabytes				
		Relational entries	Key-value pairs	JSON objects	XML objects	Nodes and edges of graph
1	10	1.2 & 1.1	25.2 & 233.7	25.2 & 219.2	25.2 & 326.5	(123.1, 338.9) & 236.6
10	40	7.4 & 6.5	234.2 & 2313.1	234.2 & 2189.8	234.2 & 3568.6	(969.3, 3208.3) & 2095.8
30	60 (3 nodes)	18.3 & 15.8	636.8 & 6367.8	636.8 & 6184.9	636.8 & 11771.31	(2674.3, 10951.5) & 6191.5

5.2   Importing Time

We import three datasets, SF1, SF10, and SF30, into ArangoDB and OrientDB using command-line utilities *arangoimp* and *oetl*. Both are executed in a single thread. Since both of them have no native XML support, we skip the XML importing test (Note that one can also convert XML objects into JSON objects, but the method is simply similar to that for JSON documents). The importing time of key-value pairs is merged into the relational model's because both DBs employ the same import method. Regarding the additional cost for supporting join operations, OrientDB needs to create inverse links between relational and JSON data using *CREATE LINK* command. In comparison, there is no such cost for ArangoDB, because once the data is imported into the system, one can perform join queries immediately.
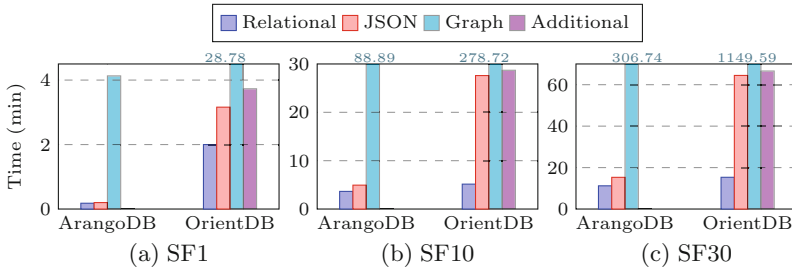
Fig. 4. Processing time for importing the multi-model datasets.

Figure 4 illustrates the result for loading three datasets. For better illustration, we measured the data loading time by four aspects, i.e., relational, JSON, graph, and additional cost. Overall, ArangoDB is 7.5x, 3.4x, and 3.8x faster than OrientDB for SF1, SF10, and SF30, respectively. Our observations are as follows. (i) For relational data, OrientDB is slightly slower as it takes time for creating unique *RID* to record the physical position for each row. On the contrary, ArangoDB employs original IDs as primary keys directly. (ii) For the JSON data, OrientDB has to transform each semi-structured JSON object into an *ODocument* object, while ArangoDB imports JSON data as *JSON lines format*, which allows it to load data in batches. (iii) For the graph data, OrientDB utilizes adjacency lists to store relations between all nodes. Thus an index lookup is needed when extracting every edge. In contrast, ArangoDB imports all edges into a *edge collection* as long as all imported documents have *_from* and *_to* attributes. This makes ArangoDB much faster than OrientDB for loading graph data. (iv) OrientDB requires additional cost for other tasks, e.g., creating links. Such cost increases drastically as data grows.

5.3   Performance of Multi-model Query

In this part, we issued ten multi-model queries on three datasets against ArangoDB and OrientDB. These queries are implemented using their query lan-

guages, i.e., *AQL* and *Orient SQL*. We use default indexes which are built on primary keys, and no secondary index is created. We provide the processing time of these queries in Fig. 5. We expect OrientDB could perform better at queries in the community level since these queries involve advanced graph traversal, but surprisingly, ArangoDB wins in most of the cases. This is due to its flexible data modeling, sophisticated query optimizer, and C++-implemented query function. Nevertheless, one exception is Q5 where OrientDB outperforms ArangoDB because the latter's query optimizer does not handle inner joins between graph and JSON efficiently, while OrientDB uses *composite SQL* queries to fetch correlated data from graph and JSON at the same time.
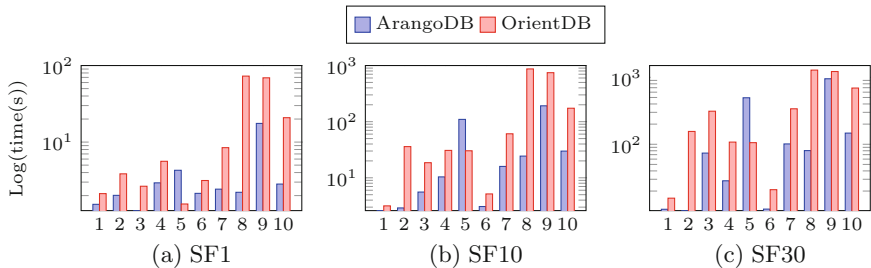

(a) SF1   (b) SF10   (c) SF30

Fig. 5. Processing time on a logarithmic scale for queries, x-axis labels are query ids, i.e., Q1 to Q10.

5.4 Transaction Performance

We adopt Java and Node.js APIs which are only feasible ways at present to implement multi-model transactions for OrientDB and ArangoDB, respectively. This leads to two different patterns: synchronous processing for OrientDB, and asynchronous processing for ArangoDB. Similar to transactional operations in RDBMS, OrientDB utilizes *begin, rollback, commit* commands to proceed transactions. However, no such commands exist in ArangoDB. Instead, it executes a transaction via an *executeTransaction* JavaScript function. All involved data in the transaction needs to be declared beforehand.

Table 4. Throughput (transactions/second) of multi-model transactions

Database	Access method	Throughput for new order	Throughput for payment
ArangoDB	Asynchronous (Nodejs)	230.6	738.5
OrientDB	Synchronous (Java)	138.3	22.9

We ran two individual transactions (i.e., *New Order* and *Payment*) with a single thread for one minute, then compute the throughput per second. The

operations of transactions in detail can be found in Table 2. Two DBs manage to *roll back* invalid transactions and *commit* valid ones, which means ACID properties on two multi-model transactions are guaranteed. Table 4 illustrates performances of both systems. The results indicate ArangoDB is better at write-heavy transaction (*Payment*) and OrientDB is more efficient in performing read-heavy transaction (*New order*). We believe this is due to the difference of their storage engines, i.e., LSM-tree-based storage for ArangoDB and B-tree-based storage for OrientDB.

6   Conclusion

Benchmarking multi-model databases is a challenging task since current public data and workloads can not well match various cases of applications. In this article, we introduce UniBench, a novel benchmark for multi-model databases. UniBench consists of a mixed data model, a scalable multi-model data generator, and a set of workloads including the multi-model aggregation, join, and transaction. Furthermore, we implement our proposed workloads on ArangoDB and OrientDB to illustrate the feasibility and usability of UniBench.

Several lessons are learned from the experimental study: (i) MMDBs are able to ingest a variety of data into storage without much additional efforts, (ii) MMDBs are able to support multi-model joins, such as graph-JSON, JSON-relational, and graph-relational. However, they lack specific algorithms to optimize the execution plan. (iii) MMDBs are able to support multi-entity and multi-model ACID transactions in the stand-alone mode, but the support for distributed ACID transactions remain on the future schedule.

As for future work, we would like to (i) introduce the flexibility into data generation because the data schema and data model in the real application could be changed dynamically, (ii) evaluate the performance of multi-model databases regarding different sharding strategies, and (iii) provide an open-source kit used to setup and run the benchmark, including the release of data generator and query implementations.

References

1. ArangoDB: Multi-model NoSQL database (2018). https://www.arangodb.com/
2. Carey, M.J., DeWitt, D.J., Naughton, J.F.: The 007 benchmark. In: ACM SIG-MOD, pp. 12–21 (1993)
3. Chen, Y., et al.: A study of SQL-on-Hadoop systems. In: Big Data Benchmarks, Performance Optimization, and Emerging Hardware, pp. 154–166 (2014)
4. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: ACM SoCC, pp. 143–154 (2010)
5. DeWitt, D.J.: The Wisconsin benchmark: past, present, and future. In: The Benchmark Handbook, pp. 119–165 (1991)

6. Erling, O., et al.: The LDBC social network benchmark: interactive workload. In: SIGMOD (2015)
7. Fader, P.S.: Customer-base analysis with discrete-time transaction data. Ph.D. thesis, University of Auckland (2004)
8. Fader, P.S., Hardie, B.G., Lee, K.L.: RFM and CLV: using ISO-value curves for customer base analysis. J. Mark. Res. **42**(4), 415–430 (2005)
9. Feinberg, D., Adrian, M., Heudecker, N., Ronthal, A.M., Palanca, T.: Gartner magic quadrant for operational database management systems, 12 October 2015
10. Ghazal, A., et al.: BigBench: towards an industry standard benchmark for big data analytics. In: ACM SIGMOD (2013)
11. Gupta, S., et al.: Modeling customer lifetime value. J. Serv. Res. **9**(2), 139–155 (2006)
12. Huang, Z., Benyoucef, M.: From e-commerce to social commerce: a close look at design features. ECRA **12**, 246–259 (2013)
13. Lehmann, J., et al.: DBPedia - a large-scale, multilingual knowledge base extracted from Wikipedia. Semant. Web **6**(2), 167–195 (2015)
14. Leskovec, J., Adamic, L.A., Huberman, B.A.: The dynamics of viral marketing. TWEB **1**(1), 5 (2007)
15. Lu, J.: Benchmarking holistic approaches to XML tree pattern query processing. In: DASFAA Workshops, pp. 170–178 (2010)
16. Lu, J.: Towards benchmarking multi-model databases. In: CIDR (2017)
17. Lu, J., Holubová, I.: Multi-model data management: what's new and what's next? In: EDBT (2017)
18. Oliveira, F.R., del Val Cura, L.M.: Performance evaluation of NoSQL multi-model data stores in polyglot persistence applications. In: IDEAS, pp. 230–235 (2016)
19. OrientDB: Multi-model & graph database. http://orientdb.com/orientdb/
20. Pluciennik, E., Zgorzalek, K.: The Multi-model databases - a review. In: BDAS, pp. 141–152 (2017)
21. Poess, M., Rabl, T., Jacobsen, H., Caufield, B.: TPC-DI: the first industry benchmark for data integration. PVLDB **7**(13), 1367–1378 (2014)
22. Prat, A., Averbuch, A.: Benchmark design for navigational pattern matching benchmarking (2015). http://ldbcouncil.org/sites/default/files/LDBC_D3.3.34.pdf
23. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: a benchmark for XML data management. In: VLDB, pp. 974–985 (2002)
24. Stonebraker, M.: The case for polystores (2015). http://wp.sigmod.org/?p=1629
25. Transaction Processing Performance Council: TPC Benchmark C (Revision 5.11) (2010)
26. Wadsworth, E.: Buy'til you die-a walkthrough (2012)
27. Zhang, K.Z.: Consumer behavior in social commerce: a literature review. Decis. Support Syst. **86**, 95–108 (2016)