Distributed and Parallel Databases (2021) 39:1–33
https://doi.org/10.1007/s10619-019-07279-6

Holistic evaluation in multi-model databases benchmarking

Chao Zhang[1,2,3] · **Jiaheng Lu**[1]

Published online: 6 December 2019

Abstract
A multi-model database (MMDB) is designed to support multiple data models against a single, integrated back-end. Examples of data models include document, graph, relational, and key-value. As more and more platforms are developed to deal with multi-model data, it has become crucial to establish a benchmark for evaluating the performance and usability of MMDBs. In this paper, we propose UniBench, a generic multi-model benchmark for a holistic evaluation of state-of-the-art MMDBs. UniBench consists of a set of mixed data models that mimics a social commerce application, which covers data models including JSON, XML, key-value, tabular, and graph. We propose a three-phase framework to simulate the real-life distributions and develop a multi-model data generator to produce the benchmarking data. Furthermore, in order to generate a comprehensive and unbiased query set, we develop an efficient algorithm to solve a new problem called multi-model parameter curation to judiciously control the query selectivity on diverse models. Finally, the extensive experiments based on the proposed benchmark were performed on four representatives of MMDBs: ArangoDB, OrientDB, AgensGraph and Spark SQL. We provide a comprehensive analysis with respect to internal data representations, multi-model query and transaction processing, and performance results for distributed execution.

Keywords Benchmarking · Multi-model · Data generation · Parameter curation

✉ Jiaheng Lu
jiaheng.lu@helsinki.fi

Chao Zhang
chao.z.zhang@helsinki.fi

1 Department of Computer Science, University of Helsinki, Helsinki, Finland

2 MOE Key Laboratory of DEKE, Renmin University of China, Beijing, China

3 School of Information, Renmin University of China, Beijing, China

1 Introduction

Multi-model dataBase (MMDB) is an emerging trend for the database management system [27,28], 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 [42] 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 [18] 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, ranging from social commerce [52], integrated health-care services [43], online recommender system [40], to smart traffic management [39].

To illustrate the challenge of multi-model data management, consider the toy example depicted in Fig. 1, which includes the customers' feedback from a table, a social network graph and the orders information in JSON files. Suppose a recommendation query for online users: *Given a customer and a product category, find this customer's friends within 3-hop friendship who have bought products in the given category, also return the feedback with the 5-rating reviews.* 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*). Note that there are three types of joins in the query: Graph-Graph ($\bowtie_a$), Graph-JSON ($\bowtie_b$) and JSON-KV ($\bowtie_c$) join. 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. Moreover, processing this query is challenging as 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.

Database benchmark became an essential tool for the evaluation and comparison of DBMSs since the advent of Wisconsin benchmark in the early 1980s. Since then, many database benchmarks have been proposed by academia and industry for various evaluation goals, such as TPC-C [44] for RDBMSs, TPC-DI [35] for data integration; OO7 benchmark [12] for object-oriented DBMSs, and XML benchmark systems [41] 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 [14] for cloud serving systems, LDBC [15] for Graph and RDF DBMSs, BigBench [13,19] 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 three challenges to be addressed when evaluating the performance of MMDBs. The first challenge is to generate synthetic multi-model data. Existing data generators cannot be directly adopted to evaluate MMDBs because they only involve one model and simulate a particular scenario. 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

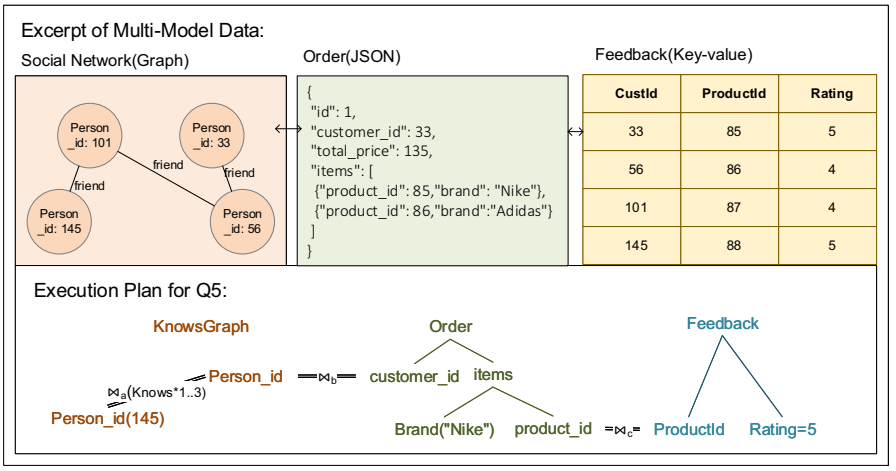


Fig. 1 Motivation example for multi-model data

to study them. The third challenge is to estimate the selectivity when generating the query. The rationale is that different parameter values for the same query template would result in different running time. Moreover, when it comes to multi-model context, it requires us to answer two interesting questions: (i) how to characterize workloads concerning the data model, and (ii) how to efficiently select parameters for holistic evaluation.

To address the first challenge, we develop a new data generator to provide correlated data in diverse data models. It simulates a *social commerce* scenario [52] that combines the social network with the E-commerce context. 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. Specifically, we propose a new probabilistic model called CLVSC to make fine-grained predictions in the third phase. For the second challenge, 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 [37], 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. To address the third challenge, we propose a vector-based approach to represent intermediate results from the multi-model perspective; then we formalize the problem as *multi-model parameter curation*. Since computing all the intermediate results of the parameter domain is costly, we develop a stratified sampling method to efficiently select the parameters for the query.
We summarize our contributions as follows:

1. We develop a new data generator, which provides correlated data in diverse data models. We propose a new framework to generate data for modeling the customers' behaviors in social commerce. We implement the generator on top of Spark SQL [8] 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 identify a new problem called *multi-model parameter curation* aiming at introducing the query diversity with curated parameters to the workloads. We then propose a sample-based algorithm to judiciously select the parameters for the benchmarking query.
4. We conduct comprehensive evaluation over four MMDBs: ArangoDB [11], OrientDB [33], AgensGraph [10], and Spark SQL [8]. We analytically report the performance comparison and our learned lessons.

This article is an extension from our previous conference version [51]. In particular, this article improves the previous version mainly in four aspects: (1) introduced a new problem of parameter curation for benchmarking the multi-model query; (2) proposed a sampling-based algorithm to select the parameters with the diversity guarantee; (3) provided more theoretical results and analysis for UniBench ; (4) conducted extensive experiments over four state-of-the-art MMDBs and give an insightful analysis.

The rest of this paper is organized as follows. Section 2 introduces the background and related work. Section 3 presents an overview of UniBench's workflow. Section 4 illustrates the data model and data generation. Section 5 presents the multi-model workload and design principles in detail. Section 6 introduces the problem of parameter curation and gives the solutions. We evaluate the data generation and parameter curation in Sect. 7. The experimental results are shown in Sect. 8. Finally, Sect. 9 concludes this work.

2 Background and related work

In this section, we briefly introduce the background and related work. In particular, Sect. 2.1 take a look back at the evolution of multi-model data management, Sect. 2.2 presents the related work to multi-model database benchmarking.

2.1 A brief introduction to MMDBs

Multi-model data management is proposed to address the "*Variety*" challenge of data since the 2010s. Generally, there are two kinds of multi-model management systems. On the one hand, many SQL-extension ecosystems, e.g., Oracle [32], PostgreSQL [36], and NoSQL systems e.g., MongoDB [30] and Redis [7], have been transformed to multi-model systems by integrating additional engines into a unified platform for supporting additional models. On the other hand, there emerge many native multi-model databases, to name a few, ArangoDB [11], AgensGraph [10], OrientDB [33]. These native systems utilize a single store to manage the multi-model data along with a unified query language. Table 1 shows the representatives of MMDBs compared by

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
PostgreSQL	SQL-extension	Relational	ALL but graph	One engine
Marklogic	Xpath	XML	JSON, RDF	One engine
MongoDB	API	JSON	Graph	One engine
Redis	API	Key-value	Graph, JSON	One engine
Spark SQL	SQL-like	DataFrame	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

several properties, namely, query language, primary model, secondary model, and storage strategy. The secondary model of each system is extended in the second evolution. For example, Redis [7] adds JSON and graph modules on top of its key-value store. Oracle [32] stores the JSON data into a column, then exploits SQL and SQL/JSON path language as the inter- and intra-document query language, respectively. Furthermore, many cloud platforms such as CosmosDB [1], Datastax [2], DynamoDB [3], employ 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. However, these kinds of MMDBs cannot support native cross-model query, that is, users have to decompose the multi-model query to the single queries, and then transform and combine the individual results in the application level. Big data systems, such as Spark [50], has been growing to provide a uniform data access to deal with multi-model data based on a distributed file system. Particularly, the Spark SQL [8] now can handle the various kinds of data, such as graph, key-value, and JSON data.

2.2 Related work

Benchmarking is a common practice for evaluating and comparing different database systems driven by database vendors and academic researchers. Nevertheless, the database benchmarks have different characteristics from the specification, data format, and workloads. Transactional benchmarks such as TPC-C [44] are typically based on business transaction-processing workloads. Analytic benchmarks such as TPC-H [45], TPC-DS [34] focus on complex queries that involve large table scans, joins and aggregation. TPC-DI [35] features a multi-model input data including XML, CSV, and textual parts, which is used to evaluate the transformation cost in the process of data integration. Bigbench [19] incorporates semi-structured (logs) and unstructured data (reviews) into TPC-DS's structured data and specifies complex queries that aim for big data analytics from the business and technique perspective. Concerning the

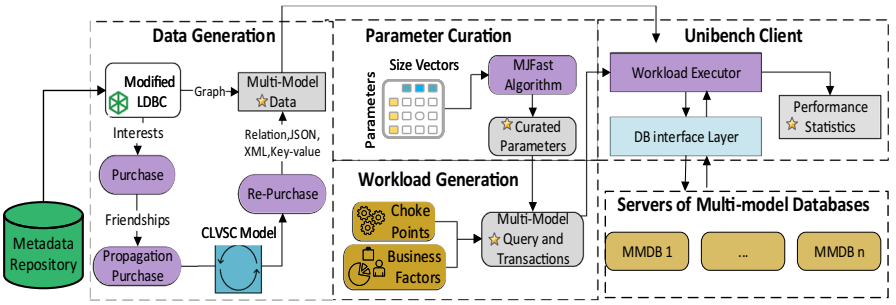


Fig. 2 Overview of our approach

NoSQL benchmarks, the YCSB benchmark [14] is designed to measure the performance of different distributed "cloud" database systems with mix workloads such as read–write operations. XMark [41] is proposed to compare the XML databases in XML query processing ranging from string matching to queries involving hierarchical path expressions. Recently, LDBC [15] and Rbench [38] benchmarks have been proposed to emphasize graph traversals and complex graph analysis, particularly on property graph and RDF data. However, none of them is suitable for benchmarking multi-model databases due to the lack of specific workloads and data. This brings a need for an end-to-end benchmark for multi-model query processing.

Our graph generation is based on LDBC [15]; we choose it as the starting point for its scalability and rich semantics in simulating social networks. The problem of *parameter curation* is proposed by [20] to control the selectivity during query generation. However, its evaluation goal is different from ours. Their approach aims for the stable runtime behaviors but our approach attempts to introduce the diversity to the query in a controllable way.

3 An overview of the benchmarking system

Figure 2 gives an overview of our benchmarking approach, which consists of three key components to evaluate the multi-model databases, including *Data Generation*, *Workload Generation*, and *Parameter Curation*. In particular, they are developed to tackle three benchmarking challenges discussed in Sect. 1.

The first component is the *data generation*; the objective is to provide multi-model, realistic, and scalable dataset. In specific, we first collect the metadata from Amazon review [26] and DBpedia dataset [25]. Then we modify the LDBC data generator [15] to generate the graph data. As our goal is concentrating on benchmarking multi-model databases rather than graph databases, we have simplified the graph complexity to better fit our goal. The original graph schema of LDBC [9] has 11 entities connected by 20 relations. The main entities are Persons, Tags, Forums, Messages (Posts, Comments, and Photos), Likes, Organizations, and Places. Our graph part consists of 3 entities connected by four relations, including entities Persons, Tags, and Post. We generate the relational, JSON, XML, and key-value data based on a Spark-based framework.

This framework consists of purchase, propagation-purchase, and re-purchase phases by considering the purchase interests, friendship, customer lifetime value, respectively. Particularly, we propose a new probabilistic model called CLVSC in the re-purchase phase to capture the information on customers' social activities. The detailed generation workflow is presented in Sect. 4.

The *workload generation* component generates the multi-model query and transactions. Particularly, we specify ten queries and two transactions for evaluating the most important aspects of multi-model query processing. The workloads are specified in English since no clear theoretical foundation, and query standard for multi-model databases has yet emerged. The fundamental design principles are the *business factors* and *choke points*. For the business factors, we identified four business levers that fit in the UniBench workload and consider various business cases such as 360-degree customer view, online recommendation, and RFM analysis. From a technical perspective, the queries are designed based on the so-called *choke points* design methodology which covers the "usual" challenges of multi-model query processing, such as point query in key-value data; graph traversal and shortest path-finding in graph; path navigation and string matching in document; join and aggregation performance in complex query. One can also find all the query definitions in AQL, Orient SQL, and SQL/Cypher for implementing Unibench in new databases. We will present the workload specification in Sect. 5.

The third key component is the *parameter curation* component. This component is responsible for curating parameters for the query to introduce the query diversity into the benchmarking process. However, the problem is non-trivial as it is computationally intensive and involves an NP-hard problem. We propose a heuristic algorithm called MJFast algorithm to tackle this problem. In particular, we first characterize the multi-model query by identifying the parameters and corresponding involved data models. Then size vectors associated with a parameter value in the base dimension are generated. Finally, the k curated parameters for the multi-model query are selected based on the Latin hypercube sampling (LHS) method. We have theoretically shown that the MJFast algorithm is superior to random sampling concerning time complexity and parameter diversity. We will discuss this component in detail in Sect. 6.

We have developed a UniBench client to load the data, execute the workloads, and collect the performance metrics, which is shown in the rightmost part of Fig. 2. This Java-based client can execute a sequential series of operations by making calls to the database interface layer. It can also control the data scale and execution threads. As matter of fact, we have utilized the UniBench client to test four MMDBs: ArangoDB [11], OrientDB [33], AgensGraph [10], and Spark SQL [8]. The benchmarking result is presented in Sect. 8.2.

4 Data model and data generation

In this section, we introduce the data model and data generation in UniBench. In particular, Sect. 4.1 presents the detailed schema and data model, Sect. 4.2 introduces the process of multi-model data generation

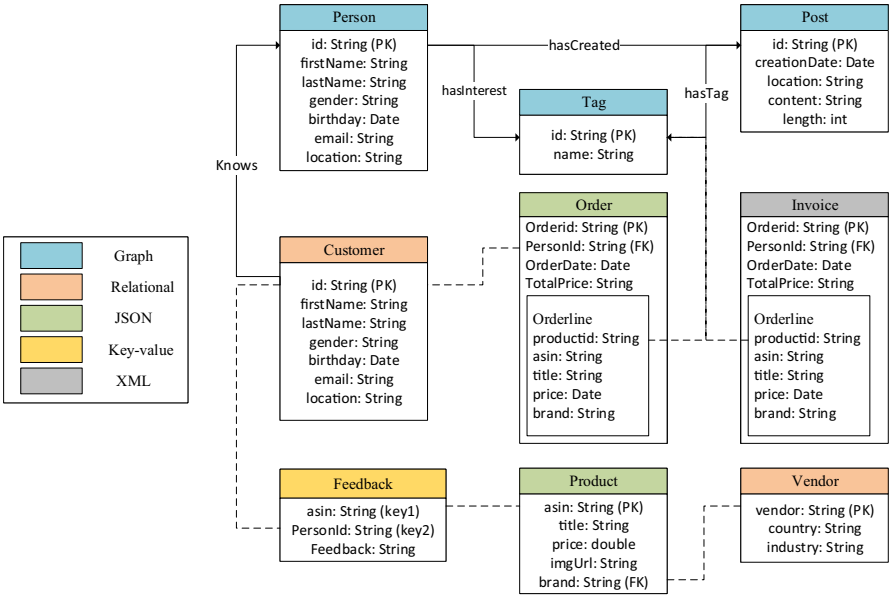


Fig. 3 Detailed schema of multi-model data

4.1 Data model

Figure 3 shows the detailed schema of our multi-model data including five data models. In particular, 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 three relations. i.e., *person, post, tag, person_hasinterest_tag, person_hascreated_post, post_hasTag_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).

4.2 Data generation

Figure 4 shows our three-phase data generation framework. Specifically, (i) in *Purchase* phase, LDBC [15] 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 CLVSC 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 detailed implementation of the algorithm can be found in the preliminary version of this paper [51]. Next, we discuss the three phases in detail as follows:

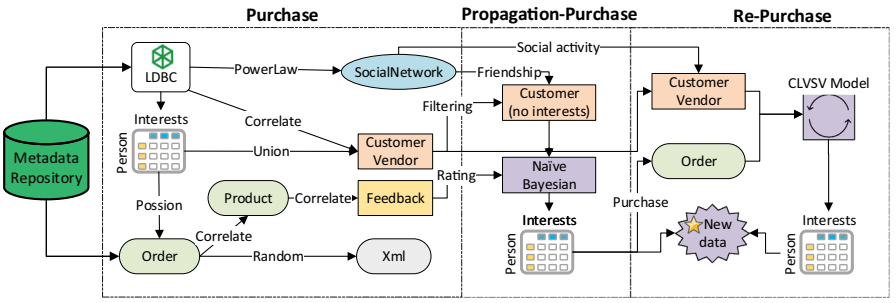


Fig. 4 Data generation workflow

4.2.1 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 interests 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).

4.2.2 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 from 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 follows:

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

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 a portion of the ranking list with the same size of interests in the purchase phase, and then generate the new transactions. If the size of the ranking list is smaller than that in the previous phase, we will randomly generate the remaining transactions.

4.2.3 Re-purchase

The CLV (customer lifetime value) model [21] 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:

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

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

Let $E(X^* \mid n^*, x', n, m, \alpha, \beta, \gamma, \delta)$ 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 $\leq$ 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) [16] 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(·) is the likelihood function. This result is derived from taking the expectation over the joint posterior distribution of active probability and inactive probability.

Following the Fader, Hardie, and Berger's approach [17] of adding monetary value, $E(M \mid p, q, \upsilon, m_x, x)$ denote the expected monetary value. Then 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 [47], 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 take a portion of the interests with the same size of interests in the purchase phase, finally generate the new transactions.

We denote the PDF for the bounded power-law (BPL) distribution with minimum value $x_{\min}$ as follow:

$$p(x) \propto \frac{\alpha - 1}{x_{\min}} \left(\frac{x}{x_{\min}} \right)^{-\alpha} \tag{6}$$

where α is the scaling parameter for the BPL distribution that typically lies in the range (2, 3), then, after the three-phase data generation, the size of person's purchase, purchase of person's friends, and person's purchases concerning the brand follow the BPL distribution with minimum value x_p, x_{pp}, and x_{rp}, respectively.

5 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 [24]: *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 [37] 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. Due to space restrictions, we only summarize their characteristics here. Additional details (e.g. the detailed description, involved data models, the input and output data) can be found in the conference version of this paper [51].

5.1 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, simulate two typical multi-model transactions for an online shopping scenario. 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.

5.2 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.

5.2.1 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 models.

5.2.2 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.

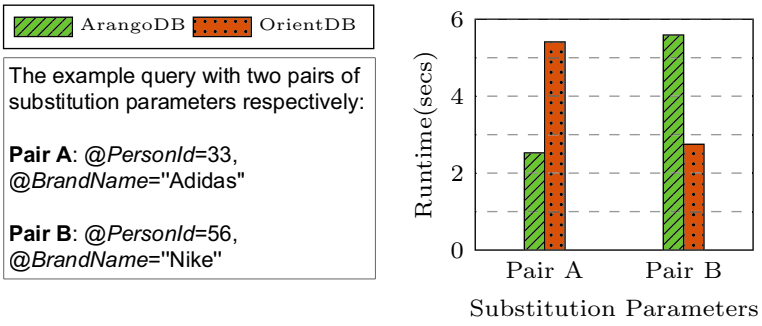


Fig. 5 Motivation of parameter curation

5.2.3 Ensuring the consistency over read–write transaction

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 of read–write transaction. The transactions not only involve read–write operations on multiple entities but also require the MMDB to guarantee the consistency across the data models. In specific, the *New Order* transaction features a mostly read query over three entities: order, product, and invoice. The *New Payment* transaction features a mostly write query over three entities: order, customer, and invoice.

6 Parameter curation

In this section, we introduce the problem of parameter curation in database benchmarking. Specifically, Sect. 6.1 presents the motivation, and Sect. 6.2 introduces the problem definition and our solution.

6.1 Motivation

A multi-model query requires configuring the *substitution parameters*, and different *substitution parameters* would result in different benchmarking performance. As shown in the left part of Fig. 5, *PersonId/56* and *BrandName/"Nike"* are two *substitution parameters* that can be replaced by other values for the query template. The experiment result in the right part of Fig. 5 illustrates that by running the query twice with two different pairs of *substitution parameters* in two multi-model databases: ArangoDB [11] and OrientDB [33], their performances are completely opposite. Interestingly, we observed this is because the same queries with different parameters differ in sizes of intermediate results. For example, the query with pair A involves relative larger intermediate results of JSON while query with pair B takes in the larger size of the graph. Therefore, an open question arises: *Can we devise an efficient and transparent method to control the query selectivity by choosing different parameter values*

for a comprehensive evaluation? In specific, we aim to introduce the query diversity to the benchmarking process by judiciously selecting the parameter values. With the query diversity, we may gain more insights from the query evaluation as Fig. 5 indicates. However, it poses three new challenges to configure parameters in benchmarking multi-model query : (i) how to represent the size of intermediate results from the data model perspective, (ii) how to select the parameters efficiently and comprehensively, and (iii) how to cover various workloads concerning the data model to introduce the query diversity to the benchmarking process. In light of this, we present a representation of involved sizes of intermediate results for a query with different parameters. We then formalize a new problem called *multi-model parameter curation* that is aimed at diversifying the query selectivity under the control of parameter curation. Due to this problem is computationally intensive, we propose a sampling algorithm to tackle this problem efficiently and effectively.

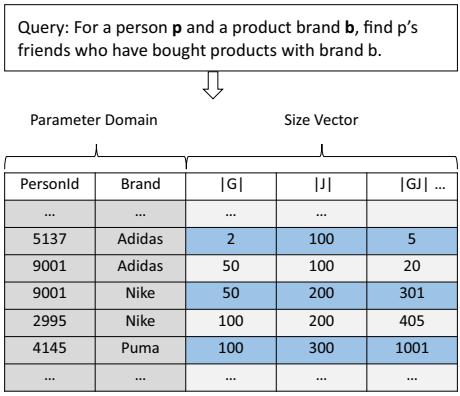
6.2 Problem definition and solutions

We propose a vector-based approach to represent the size of intermediate results associated with parameter values. Specifically, we compute sizes of all intermediate results correspond to a parameter value based on the permutation of data models. For instance, given a multi-model query: "For a person p and a product brand b, find p's friends who have bought products with brand b." The parameter domain consists of two base parameter dimensions: person p and brand b, we thus compute a non-zero vector ($|G|$, $|J|$, $|GJ|$) based on the query and parameter domain, where $|G|$ stands for the size of intermediate results from Graph, $|J|$ is size of JSON, $|GJ|$ refers to join sizes between graph and JSON models, i.e., persons who are p's friends and have bought products in brand b. This method not only allows us to represent the size of results independent of the databases, but also will help us reason out the performance results by transparently analyzing the intermediate results. Consequently, the definition of size vector is as follow:

Definition 1 *Size vector*: for a multi-model query, each size vector is defined as ω $\{c_1, ., c_k, ., c_n\}$, where c_k is k-th intermediate result size against involved data models or their combinations, c_n is the final result size. The length of ω is between $[3, 2^m - 1]$, where m is the number of the data model.

Intuitively, we aim to perform the same query with different size vectors to test the system's robustness; the more disparate the size vectors are, the more diverse the queries are. We choose the sum of min-pairwise distances of the size vectors as the diversity measure, and we use the Euclidean distance to measure the disparity. Here the diversity measure requires some explanation: our concept of diversity is a bit different from those in the field of Information Retrieval. While they aims to diversify the search results [48], our goal is to find size vectors with disparate values for the purpose of performance evaluation. Hence, our parameter curation problem boils down to finding farthest size vectors.

Now, the parameter diversity and the problem of *multi-model parameter curation* are defined as follows:



PersonId	Brand	|G|	|J|	|GJ| ...
...	...	...	...	
5137	Adidas	2	100	5
9001	Adidas	50	100	20
9001	Nike	50	200	301
2995	Nike	100	200	405
4145	Puma	100	300	1001
...	...	...	...	...

Fig. 6 Parameter curation

Definition 2 *Parameter diversity*: for a d-dimensional parameter domain P^d, each point $p \in P^d$ corresponds to a size vector ω. The distance between two ω is the normalized Euclidean distance. Given a set of points $S_k \subset P^d, k \geq 2$. The parameter diversity is defined as the sum of k size vector's minimum pair distances.

Example 1 Given a query and the corresponding parameter domain (the two leftmost columns in the table), we need to compute the size vectors depicted in Fig. 6. Suppose the required size of parameters is 3, our goal is to return three parameters groups whose size vectors are most distant. By computing the normalized Euclidean distances among all the pair-wise size vectors, here the three vectors in blue color have the maximal minimum pair distance: $0.9+0.3+0.3=1.5$, therefore we choose three parameter groups to evaluate the query: (5137, Adidas), (9001, Nike), (4145, Puma).

6.2.1 The problem of multi-model parameter curation

Given a multi-model query Q with a d-dimensional parameter domain P^d that is a Cartesian product of the base domain, as well as the size k. The objective is to select a subset $S_k \subset P^d$ such that the parameter diversity of S_k is maximal.

There are two difficulties we encounter to solve the problem of *multi-model parameter curation*. First, the parameter domain is potentially huge, which makes it infeasible to compute all the size vectors. Second, finding the k farthest points from a set of points S would need to tackle a difficult problem that is inverse to the *K-center problem* [23], which has been proved to be NP-hard. Therefore, we propose a new algorithm based on the Latin hypercube sampling (LHS) [29] to tackle this problem. LHS is a multi-dimensional stratified sampling method that can improve the coverage of the input space by independently sampling values from non-overlapping segments. Our main idea is that while it is hard to compute all the size vectors, the intermediate result sizes of the base dimensions, e.g., $|G|$ and $|J|$ are independent and can be individually computed beforehand, we then employ the LHS method with the partial knowledge to guarantee the size ranges of the base dimensions are fully represented.

Algorithm 1: Multi-model parameter curation—MJFast

Input: Parameter size k, Parameter domain $\mathbb{P}^d$, Curated query Q
Output: S_k as a subset of $\mathbb{P}$, with size k

```
1  V_k, B^d, S_k ← ∅ ;        // Initial sets of size vectors, buckets and the
   result set
2  foreach P ∈ ℙ do    // Enumerate the base dimension in parameter domain
3  |  foreach p ∈ P do
4  |  |  {s_1^p, ··· , s_n^p} ← Compute(p, Q) ;            // Compute the size vector
5  |  |  foreach i ∈ [1, k] do       // Divide the size ranges into k buckets
6  |  |  |  B_i^p ← {s_1^p, ··· , s_n^p}
7  |  |  end
8  |  end
9  end
10 B^d ← B_1 × ··· × B_d ;                  // Cartesian product of all buckets
11 foreach b⃗ ∈ B^d do
12 |  V_k ← V_k ∪ b⃗ ;                 // Sample k vectors without replacement
13 end
14 foreach v' ∈ V_k do
15 |  S_k ← S_k ∪ Lookup(D, v') ;          // Search in Dictionary by v' for S_k
16 end
17 return S_k
```

Lemma 1 *The parameter diversity between two size vectors increases as the distances of their base dimensions increase.*

Proof (*Sketch*) According to Eq. 6, the multi-model join sizes monotonically increases as the size of the base dimension increases. As the size vector merely consists of size of base dimensions and their join sizes, the Lemma 1 holds. □

The entire curation process is presented in Algorithm 1, named MJFast. Specifically, our algorithm operates in three steps: (i) line 2–3 compute the result size regarding each base dimension, (ii) line 4–9 employs the LHS method to select the size vectors, which divides the size range of each base dimension to k segments, then sample k size vectors from the d-dimensional buckets. Without loss of generality, we partition the k buckets uniformly within the range size. (iii) line 10–13 maps the size vectors to the corresponding parameter groups. Fortunately, since we are generating the data anyway, we can materialize the corresponding counts e.g., number of friends per person, number of products per brand as a by-product of data generation. This strategy will significantly speed up our process of parameter curation.

Example 2 Revisit the example of parameter curation in Fig. 6, given the curated parameter size k is 3, we first compute the base dimension $|G|$ and $|J|$ separately, then we sort and group them into three groups respectively: $|G|$: (group 1: 2, group 2: 50, group 3: 100), $|J|$: (group 1: 100, group 2: 200, group 3: 300). Next, we use the LHS method to guarantee that each group is chosen once, then we map the chosen size vectors to the parameter groups that would be the recommended parameters.

Lemma 2 *The complexity of the MJFast algorithm for the multi-model parameter curation problem is O(nlogn+n*c).*

Proof (*Sketch*) Since both the required parameter size k and involved base dimension d are very small integers, the overall time complexity of the algorithm is dominated by the costs of computing and sorting the size of the base dimension, which is $O(nlogn + n * c)$, where n is the size of the total parameter values, c is a constant that measures the average cost of computing the size for each value. □

Lemma 3 *Given a parameter size $k \geq 2$, the parameter diversity of k samples by Latin hypercube sampling is no less than the parameter diversity of k samples by random sampling with probability* $1 - \frac{(k-1)^{k-1}}{k^3}$.

Proof (*Sketch*) Let P be a parameter domain including B base dimensions, for each base dimension b with size n in B, b is divided into the k buckets with equal size n/k. Thus each value falls in a bucket with probability 1/k. For $k \geq 2$, the probability that a bucket contains only one random sample is $\binom{k}{1}\frac{1}{k}(1-\frac{1}{k})^{k-1}$. By *LHS* sampling, a bucket always contain only one sample. Combining Lemma 1 with an inequality: $\sqrt{(s_2^{B_2} - s_1^{B_1})^2} \geq \sqrt{(s_2^{B_1} - s_1^{B_1})^2}$, where s_1 and s_2 are two samples, and B_1 and B_2 are two buckets. Thus, we can derive that Lemma 3 holds. □

7 UniBench evaluation

In this section, we introduce the evaluation of UniBench. Specifically, Sect. 7.1 presents the experiments of data generation, and Sect. 7.2 show the results of parameter curation.

7.1 Data generation

For the experimental 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. The data generator is implemented on Spark 2.1.0 using scala 11.

Table 2 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 1GB and 10GB multi-model datasets in 10 and 40 min, on a single 8-core machine. In terms of scalability, we successfully generate 30G multi-model data within 60 min on our three-node cluster. The data generation is *deterministic*, which means that the size of generated datasets with a specific scale factor is always the same. In addition, as shown in Fig. 7, the percentage of data size of each model has a stable distribution concerning different scale factor.

Table 2 Characteristics of datasets

SF	Generation time (min)	Number ($\times 10^4$) and size in megabytes				
		Relational entries	Key-value pairs	JSON objects	XML objects	Nodes and edges of graph
1	10	1.2 and 1.1	25.2 and 233.7	25.2 and 219.2	25.2 and 326.5	(123.1, 338.9) and 236.6
10	40	7.4 and 6.5	234.2 and 2313.1	234.2 and 2189.8	234.2 and 3568.6	(969.3, 3208.3) and 2095.8
30	60 (3 nodes)	18.3 and 15.8	636.8 and 6367.8	636.8 and 6184.9	636.8 and 11771.3	(2674.3, 10951.5) and 6191.5

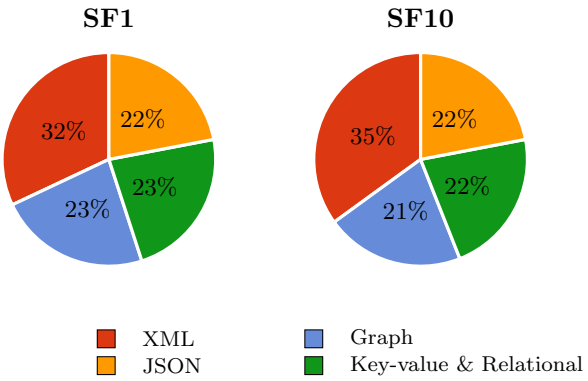


Fig. 7 Multi-model distribution of the generated dataset

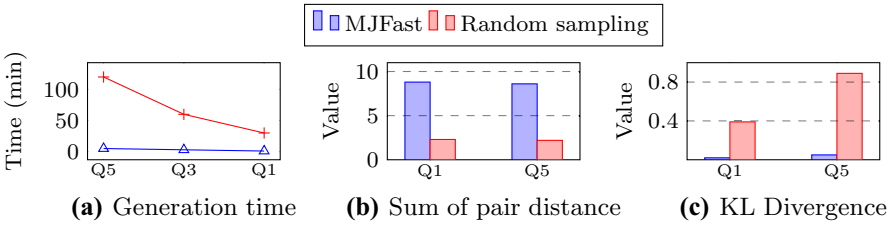


Fig. 8 Parameter curation in efficiency, diversity, and stability

7.2 Parameter curation

In this part, we compare the proposed algorithm MJFast with the random method in three aspects: generation time, the sum of pair distances, and KL-divergence D_{KL}. The experiments are conducted on a machine with a 4-core i5-4590 CPU and 16GB RAM. In particular, the generation time measures how fast the parameters can be generated. The sum of pair distances measures the parameter diversity of generated parameters. The KL-divergence D_{KL} evaluates the stability of parameter curation approaches.

As shown in Fig. 8a, our method significantly reduced the time of generating parameters for three queries. We generate the parameters for Q5, Q3, Q1 in 5, 3, 1 min, respectively. While generating all the size vectors will take approximately 2 h, 1 h and a half hour respectively. The reason that our method is much more efficient is that we only compute the result sizes for base dimensions. For the latter two aspects, we consider two cases: queries Q1 (single-parameter) and Q5 (double-parameter), respectively. Both cases are set k to 10. Figure 8b shows that parameters curated by our approach have 3× larger diversity compared to the random sampling, which verifies our theoretic analysis in Lemma 3. Finally, Fig. 8c shows our approach yields a 2× smaller D_{KL}, indicating that our approach is more robust than the random one when generating parameters multiple times.

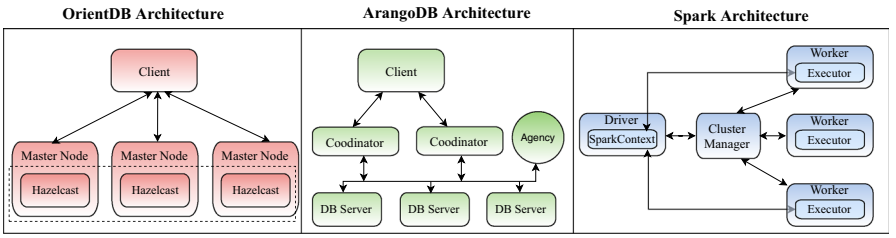


Fig. 9 The distributed architectures of OrientDB, ArangoDB, and Spark

8 System study and performance evaluation

In this section, we study and evaluate four representative multi-model databases. In Sect. 8.1, we briefly introduce the chosen systems' usability, overall implementation complexity, and distributed architecture. In Sect. 8.2, we implement all the workloads of UniBench in the tested systems and report their benchmarking results.

8.1 Systems under test

We investigate and evaluate four representatives: OrientDB, ArangoDB, AgensGraph, and Spark SQL. Figure 9 shows the distributed architectures of OrientDB, ArangoDB, and Spark, while AgensGraph is omitted as it does not support the distributed mode at the moment. In addition, we present a query implementation for each system to help the reader get the idea of how to implement the multi-model query in them using UniBench.

8.1.1 OrientDB

OrientDB [33] is an open-source multi-model database that supports multi-model query, transactions, sharding, and replication. Its core model is the graph model. In OrientDB, a record is the smallest unit that can be loaded from and stored into the database. It can be a document, a bytes record (BLOB) and a vertex, as well as an edge. Records are defined by classes that can be schema-less, schema-full, or a mix. Moreover, OrientDB implements an index-free adjacency structure to traverse the graphs without any index lookup. As a consequence, it implements a data model integrating document, graph, object, and key-value models. When it comes to the query language, it provides a unified query language called *Orientdb SQL* to query the multi-model data. Nevertheless, it does not completely follow the standard SQL syntax like SQL ANSI-92 although it is a SQL-like language. For instance, Fig. 10 shows an OrientDB SQL implementation for the motivation example, in which (i) JOIN syntax is not supported while it uses a chain of path-oriented dot operations to link the data. e.g., *persons.feedback* in line 1. (ii) it provides a TRAVERSE keyword to traverse the graph. i.e., line 3–5. (ii) it adopts a *select-from-where* SQL syntax, but the sub-SQL clause is embedded in the *From* instead of *Where* keyword. (iii) To access

```
1  SELECT person, person.feedback
2  FROM
3      (TRAVERSE Expand(Out('Knows') )
4       FROM person
5       WHERE PersonId=56 and $depth<3)
6  WHERE "Nike" in Order.items.brand and feedback.Rating==5
7  UNWIND Order.items
```

Fig. 10 Multi-model query implementation using OrientDB SQL

the embedded JSON document, it has to flatten it to multiple single-row documents. e.g., UNWIND operator in line 7.

As shown in the leftmost part of Fig. 9, OrientDB has a multi-master distributed architecture where each server in the cluster can become an on-line master node that supports both read and write. It is implemented based on the Hazelcast framework [5] that can automatically manage the cluster's configuration, communication, and fail-over, as well as the synchronization among the servers. OrientDB supports sharding the data by a designated attribute, but this can only be done manually. Nevertheless, OrientDB automatically selects the local servers to store the shards after sharding the data. Particularly, user can configure the parameter *writeQuorum* to decide how many master nodes should execute the writes in parallel, afterward, the remaining unaligned servers perform the lazy-updates.

8.1.2 ArangoDB

ArangoDB [11] is an open-source multi-model database that supports multi-model query, transactions, sharding, and replication. It provides a new query language. From the data model perspective, ArangoDB is originally a document-oriented database that is implemented on a key-value storage engine, *RocksDB*. The documents in ArangoDB follows the JSON format, and they are organized and grouped into collections, each collection can be seen as a unique type of data in the databases. The graph data model is implemented by storing a JSON document for each vertex and a JSON document for each edge. The edges are kept in special edge collections that ensure that every edge has *from* and *to* attributes which reference the starting and ending vertices of the edge. ArangoDB develop a unified and declarative ArangoDB Query Language (*AQL*) to manipulate the multi-model data. In particular, the basic building block of AQL is inspired by (*For, Let, Where, Return*) FLWR expressions from XQuery in XML except that the *Where* is replaced with *Filter*. AQL enables operations on documents such as selection, filtering, projections, aggregation, and joining. It also supports graph traversal using AQL traversal syntax which enables the graph traversal and pattern matching in a property graph, such as *neighbors*, *shortestpath* etc. More interestingly, operations for all three models, i.e., key-value, document, and graph, can be integrated into a single AQL query and be performed in databases as a whole. This feature enables the join and union of correlated data from different sources simultaneously. Figure 11 depicts a AQL implementation for the motivation example. In specific, the line 1–3 operates on the graph, JSON, and key-value data, and the filtering conditions and

```
1  FOR friend IN 1..3 OUTBOUND PersonId/56 KnowsGraph
2  FOR order IN Order
3  FOR feedback IN Feedback
4  FILTER order.customer_id==friend._id AND
5  BrandName/"Nike" IN order.items[*].brand AND
6  friend._id==feedback.custID AND
7  feedback.Rating==5
8  RETURN {person:friend, feedback:feedback}
```

Fig. 11 Multi-model query implementation using ArangoDB AQL

equijoin predicates are specified in the *Filter* clause accordingly during line 4–7. In line 5, it uses the *[*]* operator to access to the *brand* element in *items* array.

ArangoDB has a multi-role distributed architecture, which consists of three instances playing different roles in the cluster. Namely, the agent, coordinator, and database server. Specifically, the agent instance is the central place used to store the cluster configuration and provide the synchronization service based on the Raft Consensus algorithm [31]. A coordinator is a stateless instance that connects the clients and the ArangoDB cluster. It coordinates the query tasks and assigns these tasks to the local servers where the data is stored. The database servers host shards of data and can execute the incoming queries in part or as a whole. One of the salient features of ArangoDB is the auto-sharding, which can automatically distribute the data in shards to the different servers based on the hash value of the *_key* attribute, but the user can also designate the attributes for sharding the data.

8.1.3 AgensGraph

AgensGraph [10] is another open-source multi-model database that implements the graph model in the core engine of PostgreSQL. Consequently, it can support the graph, relational, key-value and document model in one core. A major advantage is that users can not only take full advantage of the relational database with SQL, but also manage the graphs and documents simultaneously. As for query language, it utilizes a hybrid query to manipulate the data without reinventing a brand-new query language. The query combines the SQL/JSON syntax and the graph query language Cypher (*match-where-return* syntax) which can be mutually compatible. In the case of the internal representation, AgensGraph implements the graph model by storing two vertex sets to two tables, and using an edge table that contains the start and end foreign key to link two tables. This storage strategy of the graph is similar to ArangoDB that also stores the vertices and edges separately. In addition, the internal vertex is represented by a binary JSON format—JSONB, meaning that it can also manage the document type. Figure 12 shows a implementation of motivation example using AgensGraph SQL. Interestingly, this SQL query contains two multi-model inner joins. The first inner join is between the relational table and items in the JSON array (line 2–5); the second join is between the intermediate result with the graph data (line 6–9). In line 3, it uses the *jsonb_array_elements* function to retrieve the items array in JSON order. In line 7–8, it leverages the Cypher syntax to traverse a knows graph.

```
SELECT person, feedback
FROM orders,
  jsonb_array_elements(orders.data->'items') element
INNER JOIN feedback
ON feedback.asin=element->>'asin'
INNER JOIN
  (MATCH(c:customers {id:'56'})-[:KNOWS*1..3]->(person:persons)
   RETURN person)
ON person->>'id'=feedback.personid;
```

Fig. 12 Multi-model query implementation using AgensGraph SQL

```
1  val persons = sqlContext.read.format("csv")
2                  .load("HDFS://person.csv").toDF()
3  val orders   = sqlContext.read.format("json")
4                  .load("HDFS://order.json").toDF()
5  val knows    = sqlContext.read.format("csv")
6                  .load("HDFS://knows.csv").toDF("src", "dst")
7  val graph    = GraphFrame(persons, knows)
8  val friends  = graph.find("(a)-[e1]->(b);(b)-[e2]->(c)")
9                  .filter("a.id=56")
10                 .select(explode(array("a.id", "b.id"))
11                 .alias("PersonId")).distinct
12 val orders=orders.where(array_contains(col("items.brand"),"Nike"))
13 val result =   orders.join(friends,Seq("PersonId"),"inner")
14                 .select("PersonId","items").collect()
```

Fig. 13 Multi-model query implementation using Spark SQL

8.1.4 Apache spark

Apache spark [50] is a very popular open-source distributed computing framework for large scale data analytics, which uses resilient distributed datasets (RDD) [49] to cache intermediate data across a set of nodes. Although spark was designed to work with structured data, it has been growing to provide a uniform data access to deal with multi-model data. Particularly, the Spark SQL module [8] now can handle the various kinds of data, such as graph, key-value, and JSON data. Spark SQL uses the *DataFrame* as the data abstraction, which borrows the concept of DataFrame from the pandas project [6]. Nevertheless, *DataFrame* in Spark SQL strongly relies on the immutable, in-memory, distributed, and parallel capabilities of RDD. Figure 13 shows an abridged implementation of the join task in Sect. 8.2.5 using Spark SQL. Specifically, line 1 to 4 load the csv and JSON data from the HDFS. In line 5–6, the knows DataFrame specifies the column names *src* and *dst* referencing the starting and ending vertices of an edge in the graph. Line 7 builds a graph via the GraphFrame package [4] which provides a DataFrame-based graph. Line 8 uses a *motif finding* operation to find the two-hop friends. Line 9 to 11 return the distinct persons among the friends. Line 12 finds the JSON orders with Nike brand, and line 13 performs an inner join between the orders and friends.

Spark follows a master/worker architecture. There is a *SparkContext* object in the driver program that allows users to configure the common properties, e.g., master URL, CPU cores, and working memory. It also can schedule the jobs to be executed in the workers. The default scheduling mode is FIFO (first-in-first-out). As shown in the rightmost part of Fig. 9, the *SparkContext* connects to a single coordinator called cluster manager that could be a spark master, Mesos [22], or YARN [46]. The cluster manager allocates resources across worker nodes in the cluster. Each worker nodes can have one or more executors that run computations and store data for the assigned tasks. Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads.

8.2 Benchmarking results

In this section, we report our experimental results over the chosen SUT. We evaluate the performance of the implemented benchmark over the generated datasets. In particular, the benchmark tasks includes the data ingest, multi-model query processing and transaction processing, and the distributed execution.

8.2.1 Setup

For the experimental setup, we conduct the benchmark experiments in the stand-alone mode on a 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 three representative MMDBs: OrientDB, AgensGraph, and ArangoDB with community version v2.2.16, v2.0.0 and v3.3.20, respectively. On the client-side, we develop a client Java program integrated with each DB's official driver. All benchmark workloads are implemented in the program. We conducted the distributed experiments on a cluster of three Amazon EC2 instances. We select three representative multi-model database clusters: OrientDB, ArangoDB, and Spark. The tested DB servers are deployed on t3.large instances with 2 vCPUs and 8 GB RAM.

8.2.2 Data ingest

The results for loading three datasets are shown in Fig. 14 (resp. processing time in log scale). For a fair comparison, we executed all the import utilities with a single thread (we found both OrientDB and Agensgraph are unable to support parallel importing). For better illustration, we separated the data loading time into four aspects, i.e., relational and key-value, JSON, graph, and additional cost, which are shown as a stacked bar in the graphs. (Note that the importing of XML is omitted because both ArangoDB and OrientDB cannot support XML natively).

Result and discussion The most striking feature of the results for the load time is the difference in the performance of OrientDB compared to AgensGraph and ArangoDB. Despite shutting down the service and load the data locally, the load times are surprisingly high (roughly 20 h for importing dataset with SF30), and the overhead for creating links increases drastically as data grows (approximately 8, 217, 1139 min for

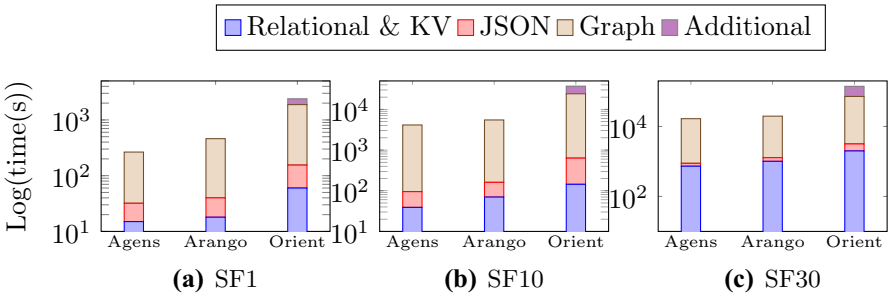


Fig. 14 Processing time for importing the multi-model datasets with single thread

three datasets, respectively). This is because: (i) for relational data, it takes time for creating unique *RID* to record the physical position for each row. (ii) For the JSON data, OrientDB has to transform each semi-structured JSON object into an *ODocument* object. (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.

Overall, AgensGraph outperforms both OrientDB and ArangoDB in a single-thread setting. This is mainly because the multi-model data can be directly imported from the data files to relational tables by bulk loading (we use a micro-batch of 10,000 records). Furthermore, the performance of ArangoDB is comparable as it also loads either CSV tables or JSON files to documents in batches without having any additional cost. In addition, if we load the data into ArangoDB with multiple threads (typically more than four threads), ArangoDB is the winner in the task of data importing. In particular, utilizing sixteen threads to load the data corresponds to performance improvement by 4.8×, 4.9×, 5.2× for three datasets, respectively.

8.2.3 End-to-end query performance

In this section, we first evaluate the performance of multi-model queries end-to-end, then we collect the statistical information of resource utilized by the tested systems. We perform the query with the same curated parameters against the systems to ensure the same result of each run. We use default indexes which are built on primary keys, and no secondary index is created. We ask three questions for the evaluation: (i) How does the performance compare across the systems? (ii) How well do the systems scale as the input data increases? (iii) What is the resource utilization of the systems when executing the queries?

Result and discussion Figure 15 shows the runtimes in all queries varying the input data size. We implement all the queries defined in UniBench using their query languages. Interestingly, the results show that the systems under test (SUT) achieve disparate performance with respect to four business categories in the workloads. For the two point queries on multi-model data, namely, Q1 in the *Individual* use case and Q2 in the *Conversation* use case, while ArangoDB and OrientDB achieve comparable performance, AgensGraph is much slower. We found that AgensGraph can respond to the individual point queries very fast, but when processing the multi-model point query, it gets much slower. This is primarily due to the separate relational store and graph store in AgensGraph, which incurs the extra overhead of passing

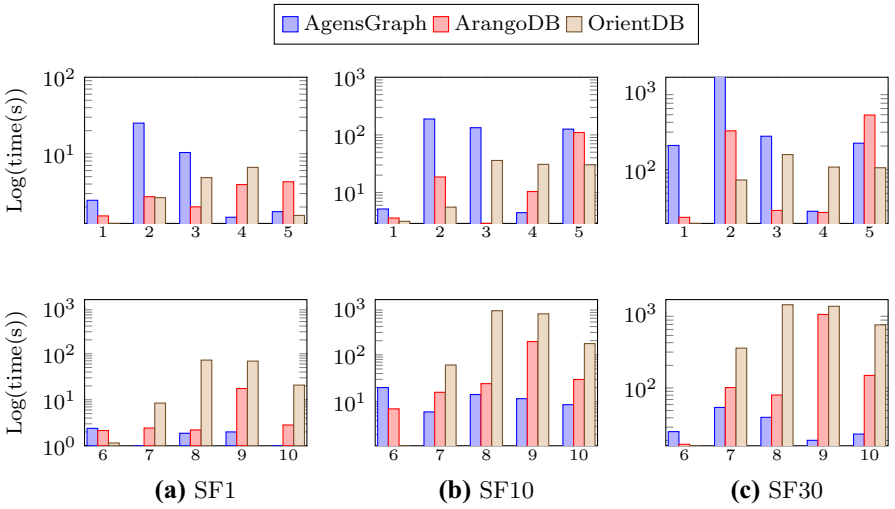
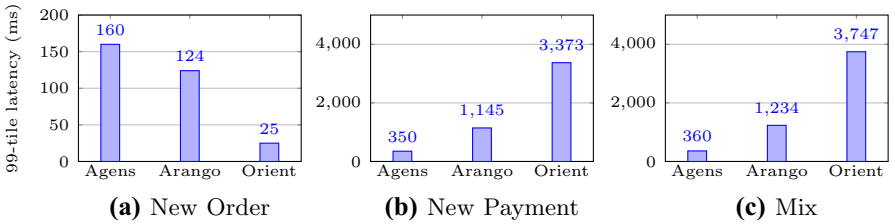


Fig. 15 Processing time on a logarithmic scale for queries, x-axis labels are query ids, i.e., Q1 to Q10

the individual query results to the final results. ArangoDB is an order of magnitude faster than the other systems in Q3, which joins the JSON orders with a post graph, along with a filter on the embedded JSON array. Thanks to the native document store and advanced array operator, ArangoDB can perform the Q3 natively and efficiently. OrientDB is slower because it has to flatten the embedded structure in order to do the filtering, leading to extra overhead. A surprising observation is about AgensGraph, which uses the *cross join* to flatten the JSON array in the case of Q3, leading to large overheads because it enumerates the orders with all the flattened JSON arrays.

For the *Community* use case (Q4–Q6), our first observation is that the operation order in these queries significantly affects the performance. In the case of Q4 that starts with a document aggregation and then graph traversal, AgensGraph and ArangoDB do better than OrientDB. In the case of Q5 and Q6, which begin with the graph traversal, then perform the document aggregation, OrientDB shows superiority. This means that the performance of multi-model join depends on their main models. i.e., AgensGraph is originally a relational database, ArangoDB is originally a document-oriented database, and OrientDB is a native graph database. Another observation is that the level of graph traversal affects the performance of AgensGraph and ArangoDB, when the level is greater than 3, they are much slower than OrientDB. This is because both of them use recursively join in traversing a graph, which is not suitable for deep graph traversal. The results for the *Commerce* use case clearly demonstrates that AgensGraph outperforms ArangoDB and OrientDB. For dataset SF30, it is approximately 9× and 25× faster than ArangoDB and OrientDB, respectively. We attribute this to its tabular-structured data layout, which is more efficient for *Group By* clause. OrientDB is the slowest to execute these analytical queries although it also provides the *Group By* clause, indicating that graph store is poor at complex aggregations due to the overhead of the duplicate linkages.



(a) New Order **(b)** New Payment **(c)** Mix

Fig. 16 Transaction performance

8.2.4 Multi-model ACID transaction performance

We ran two individual transactions (i.e., *New Order* and *New Payment*), and a mixed transaction combining them on the SUT. The operations of transactions in detail can be found in [51]. Three DBs manage to *roll back* invalid transactions and *commit* valid ones, which means ACID properties on two multi-model transactions are guaranteed. Figure 16 illustrates the 99 percentile latency of the transaction performances. The results show that AgensGraph and ArangoDB are better at the write-heavy transaction (*New Payment*) and OrientDB is more efficient in performing the read-heavy transaction (*New order*).

8.2.5 Distributed multi-model query processing

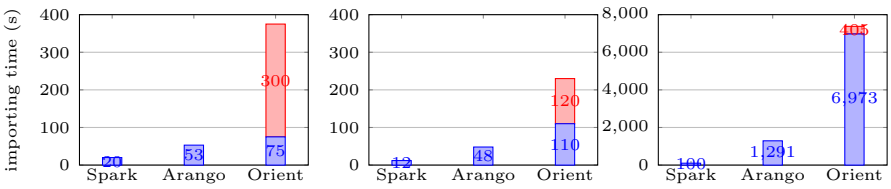
We conducted the distributed experiments on a cluster of three Amazon EC2 instances. We used the dataset with scale factor SF10. The tested DB servers are deployed on t3.large instances with 2 vCPUs and 8 GB RAM. We added the latest version of Spark (v2.4.3) to compare with two distributed multi-model databases, namely, ArangoDB (v3.4) and OrientDB (v3.0.22). In particular, each worker in the three-node Spark cluster has 1 GB working RAM and 1 executor by default. We utilized HDFS as the storage for spark in which the replication factor was set to 3, and we used Spark SQL to implement the tested tasks, the number of shuffle partitions *spark.sql.shuffle.partitions* was configured with the default value 200. We disabled the result cache in all systems for a fair comparison.

We developed three tasks related to multi-model distributed query processing, which includes the selection, join, and aggregation tasks. These three tasks are derived from the original Unibench query Q1, Q5, and Q8, respectively. We describe them in detail in the following.

Selection task: this task is to find the associated data from various data sources given a person id @id. The query in Orient SQL is as follows:

```
SELECT profile, order, feedback FROM Customer where id=@id;
```

Join task: this task consists of joins among three multi-model entities (Relational *person*, Graph *knows*, and JSON *order*). We have discussed its implementations in Sect. 8.1, but we reproduce the implementation in Orient SQL with a litter modification here for convenience:



(a) Tabular data **(b)** JSON data **(c)** Graph data

Fig. 17 Processing time for importing the multi-model datasets

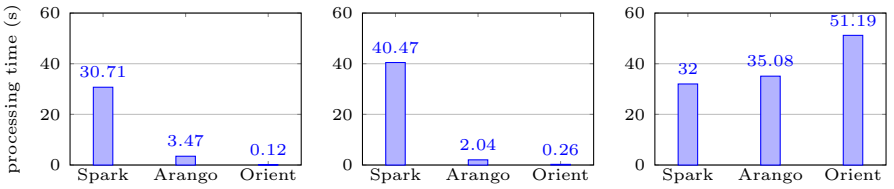
```
SELECT person, order.items
FROM (TRAVERSE ('Knows') FROM person WHERE id=@id and
$depth<3)
WHERE @brand in order.items.brand
```

Aggregation task: this task consists of two sub-tasks that perform a complex aggregation on three entities (*product*, *PostHasTag*, and *order*). The first part of the task is to find the top-10 bought items that have the highest total quantity. Then the second task is to calculate their popularity in the social media by counting the related posts.

```
SELECT items, count(items)
FROM (SELECT * FROM order) UNWIND order.items)
GROUP BY items.id
ORDER BY count DESC limit 10;
SELECT In('PostHasTag').size() as popularity
FROM Product WHERE id in items.id
ORDER BY popularity DESC;
```

Result and discussion Figure 17 demonstrates the loading time for importing the tabular, JSON and graph data, respectively. The results show that Spark is roughly 10x faster than ArangoDB and 50x faster than OrientDB. This is because in Spark, the master node simply copies each data file from the local disk into the HDFS instance and then distributes the replicas to other slave nodes in the cluster. OrientDB has a high overhead in data importing, which is shown as the red upper segments of the stacked bar. In addition to the overhead such as link creation, we found that OrientDB loaded the data to its cluster in a sequential manner, i.e., node by node. Thus, as the node size increases, the load times will increase proportionately. In contrast, the data importing in ArangoDB cluster can be done in parallel across nodes.

The processing time for performing the query tasks are illustrated in Fig. 18. For the selection and join tasks, Spark is approximately 13x slower than ArangoDB and much slower than OrientDB by three orders of magnitude. The reason has two folds. First, Spark has to scan the entire HDFS files to build the RDDs and then perform the query due to the lack of index support. While both MMDBs have the tailored storage and the default indices after the importing process. For instance, ArangoDB has the binary JSON store with the primary index on each document. OrientDB has the paginated local storage in disk with physical pointer RIDs. These structures enable both MMDBs to significantly speed up the query execution by using index scan and join without a complete scan of the datasets. Second, to perform a complex join task,



(a) Selection task **(b)** Join task **(c)** Aggregation task

Fig. 18 Distributed multi-model query processing time

e.g., the join between the two-hop friends in the knows graph and the orders, both of MMDBs are able to do the join locally on each node, (particularly for OrientDB, each node has the whole dataset in the replica mode; thus the join is completely done in one node). While in Spark SQL, the shuffle join can not be avoided because one cannot manually control the partitioner for the data locality access. As for the aggregation task, despite the overhead for loading the data, Spark slightly outperforms ArangoDB and OrientDB since Spark SQL can take advantage of multiple RDD partitions to speed up the *group by* aggregation.

8.2.6 Summary of performance evaluation

Overall, our experimental results show that (i) for data importing, AgensGraph is approximately 1.5×, 9× faster than ArangoDB and OrientDB in the single-thread setting for SF30 input, respectively. The main bottleneck of OrientDB is the creation of graph links that spends roughly 20 h to ingest the data. In addition, ArangoDB can significantly accelerate the data importing by 5× in 16 threads. (ii) For query processing, OrientDB is excel at point queries (Q1 and Q2) and path-orient queries (Q5 and Q6). Specifically, OrientDB is averagely 3.6×, 36× faster than ArangoDB and AgensGraph for SF30 input regarding the point queries. The main bottleneck of AgensGraph for point query is the retrieval of the graph and relational data without joining. Concerning the path-orient queries, OrientDB is 2×, 4× faster than AgensGraph and ArangoDB. ArangoDB is the best at document filtering and joining query (Q3 and Q4), which are averagely 4.5×, 5× faster than OrientDB and AgensGraph for SF30 input. AgensGraph outperforms the others in performing complex aggregation queries (Q7–Q10), which are averagely 9×, 25× faster than ArangoDB and OrientDB for SF30 input, respectively. (iii) For transaction processing, AgensGraph is roughly 3.4×, 10× faster than ArangoDB and OrientDB when running the two transactions together.

Regarding the distributed multi-model query processing, we observe that (i) spark is roughly 10× faster than ArangoDB and 50× faster than OrientDB in data importing. (ii) For the selection and join tasks, OrientDB is the winner, which is roughly 13× faster than ArangoDB and 187× faster than Spark SQL, respectively. (iii) For the aggregation task, Spark SQL is 1.6× faster than OrientDB by using multiple partitions; the performance of ArangoDB is comparable to Spark SQL.

8.3 Discussion

Through detailed experiments, we verified the feasibility and applicability of UniBench. The proposed benchmark UniBench also demonstrates many advantages in benchmarking multi-model databases. First, the rich data formats and skewed distributions provided by UniBench give us an opportunity to simulate the complex, realistic scenario. Even if a database may not support all the data types in UniBench, it still can implement UniBench for partial evaluation. This indicates that UniBench is a generic benchmark being able to support a wide range of databases that range from SQL/NoSQL, multi-model databases to big data systems. Essentially, all the databases in Table 1 can be implemented in UniBench with or without data transformation. Since there is no query language standard, one can find all the query definitions in AQL, Orient SQL and SQL/Cypher here,[1] where users can also implement the UniBench in the multi-model database of interest, or extend the core workloads. Moreover, the sophisticated workloads help us identify several bottlenecks of tested databases when evaluating query processing. Such insights can hardly be gained by other benchmarks in that they lack the complex and meaningful multi-model workloads. Last but not least, the parameter curation ensures that our query evaluation is holistic and robust. In summary, the detailed analysis with UniBench would be of particular interest to developers of core engines, system administrators/DBA, and researchers of multi-model databases.

We also have the following key observations from the comparative evaluation: (i) in addition to the index-free adjacency structure, relational store, and document store can also implement a Labeled Property Graph (LPG) efficiently. This is verified by the query performance of AgensGraph and ArangoDB. (ii) The tested MMDBs can support multi-model joins, such as graph-JSON, JSON-relational, and graph-relational. However, they lack specific algorithms to optimize the execution plan. (iii) The tested MMDBs can support multi-entity and multi-model ACID transactions in the stand-alone mode, but they cannot support distributed ACID transactions. (iv) The data locality access of ArangoDB and OrientDB are more flexible than Spark as they can employ the sharding function to partition the data with a global index support.

9 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 propose an efficient algorithm for parameter curation to judiciously configure parameters in benchmarking queries. We conducted extensive experiments over four representatives using UniBench.

[1] http://github.com/HY-UDBMS/UniBench.

We believe that current systems provide good support for multi-model data management, but they also open new opportunities for further improvement and future research. Exciting follow-up researches are (i) to introduce the flexibility into data generation because the data schema and data model in the real application could be changed dynamically, (ii) to evaluate the performance of multi-model databases regarding different sharding strategies, and (iii) to evaluate the performance of multi-model databases regarding different system's query optimization, e.g., execution plan optimizer, indexes, result cache.

Acknowledgements Open access funding provided by University of Helsinki including Helsinki University Central Hospital. This work is supported by Academy of Finland (310321) and China Scholarship Council fellowship (201606360137).

References

1. CosmosDB: a globally distributed, multi-model database (2019). https://docs.microsoft.com/en-us/azure/cosmos-db/
2. DataStax: active everywhere, every cloud (2019). https://www.datastax.com/
3. DynamoDB: fast and flexible NoSQL database service for any scale (2019). https://aws.amazon.com/dynamodb/
4. GraphFrames: a DataFrame-based graph package for Spark (2019). https://graphframes.github.io/graphframes
5. Hazelcast open source project (2019). https://hazelcast.com
6. Pandas: powerful Python data analysis toolkit (2019). https://pandas.pydata.org/pandas-docs/stable/
7. Redis: a distributed, in-memory key-value database (2019). https://redis.io
8. Spark SQL: a Spark module for structured data processing (2019). https://spark.apache.org/sql/
9. The LDBC Social Network Benchmark (version 0.3.2) (2019). http://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf
10. Agensgraph: Online Transactional Multi-model Graph Database (2018). http://www.agensgraph.com/
11. ArangoDB: Multi-model NoSQL Database (2018). https://www.arangodb.com/
12. Carey, M.J., DeWitt, D.J., Naughton, J.F.: The oo7 benchmark. In: Proceedings of the ACM SIGMOD, pp. 12–21 (1993)
13. Chen, Y., Qin, X., Bian, H., Chen, J., Dong, Z., Du, X., Gao, Y., Liu, D., Lu, J., Zhang, H.: A study of sql-on-hadoop systems. In: Big Data Benchmarks, Performance Optimization, and Emerging Hardware, pp. 154–166 (2014)
14. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the ACM SoCC, pp. 143–154 (2010)
15. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat-Pérez, A., Pham, M., Boncz, P.A.: The LDBC social network benchmark: interactive workload. In: Proceedings of the SIGMOD (2015)
16. Fader, P.S.: Customer-base analysis with discrete-time transaction data. Ph.D. thesis, University of Auckland (2004)
17. 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)

18. Feinberg, D., Adrian, M., Heudecker, N., Ronthal, A.M., Palanca, T.: Gartner magic quadrant for operational database management systems, 12 October (2015)
19. Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., Jacobsen, H.: BigBench: towards an industry standard benchmark for big data analytics. In: Proceedings of the ACM SIGMOD (2013)
20. Gubichev, A., Boncz, P.: Parameter curation for benchmark queries. In: Proceedings of the TPCTC, pp. 113–129 (2014)
21. Gupta, S., Hanssens, D., Hardie, B., Kahn, W., Kumar, V., Lin, N., Ravishanker, N., Sriram, S.: Modeling customer lifetime value. J. Serv. Res. **9**(2), 139–155 (2006)
22. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R.H., Shenker, S., Stoica, I.: Mesos: a platform for fine-grained resource sharing in the data center. NSDI **11**, 22–22 (2011)
23. Hochbaum, D.: The K-centre problem. In: Approximation Algorithms for NP-Hard Problems, pp. 495–523. PWS Publishing Company, Boston (1997)
24. Huang, Z., Benyoucef, M.: From e-commerce to social commerce: a close look at design features. ECRA **12**, 246–259 (2013)
25. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: Dbpedia—a large-scale, multilingual knowledge base extracted from wikipedia. Semant. Web **6**(2), 167–195 (2015)
26. Leskovec, J., Adamic, L.A., Huberman, B.A.: The dynamics of viral marketing. TWEB **1**(1), 5 (2007)
27. Lu, J.: Towards benchmarking multi-model databases. In: Proceedings of the CIDR (2017)
28. Lu, J., Holubová, I.: Multi-model data management: what's new and what's next? In: Proceedings of the EDBT (2017)
29. McKay, M.D., Beckman, R.J., Conover, W.J.: A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. Technometrics **42**(1), 55–61 (2000)
30. MongoDB: GraphLookup: performs a recursive search on a collection. (2018). https://docs.mongodb.com/manual/reference/operator/aggregation/graphLookup/index.html
31. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14), pp. 305–319 (2014)
32. Oracle: Database JSON Developer's Guide—Part II (2018). https://docs.oracle.com/database/122/ADJSN/store-and-manage-json-data.htm
33. OrientDB: Multi-model & Graph Database. http://orientdb.com/orientdb/
34. Poess, M., Rabl, T., Jacobsen, H.: Analysis of TPC-DS: the first standard benchmark for sql-based big data systems. In: Proceedings of the SoCC, pp. 573–585 (2017)
35. Poess, M., Rabl, T., Jacobsen, H., Caufield, B.: TPC-DI: the first industry benchmark for data integration. PVLDB **7**(13), 1367–1378 (2014)
36. PostgreSQL: The Official Site for PostgreSQL, the World's Most Advanced Open Source Database (2016). https://www.postgresql.org/
37. Prat, A., Averbuch, A.: Benchmark design for navigational pattern matching benchmarking (2015). http://ldbcouncil.org/sites/default/files/LDBC_D3.3.34.pdf
38. Qiao, S., Özsoyoglu, Z.M.: Rbench: Application-specific RDF benchmarking. In: Proceedings of the ACM SIGMOD (2015)
39. Rehena, Z., Janssen, M.: Towards a framework for context-aware intelligent traffic management system in smart cities. In: Proceedings of the WWW, pp. 893–898 (2018)
40. Sarwat, M., Moraffah, R., Mokbel, M.F., Avery, J.L.: Database system support for personalized recommendation applications. In: Proceedings of the ICDE, pp. 1320–1331 (2017)
41. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A benchmark for XML data management. In: Proceedings of the VLDB, pp. 974–985 (2002)
42. Stonebraker, M.: The Case for Polystores (2015). http://wp.sigmod.org/?p=1629
43. Tello-Leal, E., Villarreal, P.D., Chiotti, O., Rios-Alvarado, A.B., López-Arévalo, I.: A technological solution to provide integrated and process-oriented care services in healthcare organizations. IEEE Trans. Ind. Inf. **12**(4), 1508–1518 (2016)
44. Transaction Processing Performance Council: TPC Benchmark C (Revision 5.11) (2010)
45. Transaction Processing Performance Council: TPC-H (Revision 2.17.3) (2017). http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf
46. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al.: Apache hadoop yarn: yet another resource negotiator. In: Proceedings of the 4th annual Symposium on Cloud Computing. p. 5. ACM (2013)
47. Wadsworth, E.: Buy'til you die-a walkthrough (2012)

48. Wang, X., Wen, J.R., Dou, Z., Sakai, T., Zhang, R.: Search result diversity evaluation based on intent hierarchies. IEEE Trans. Knowl. Data Eng. **30**(1), 156–169 (2017)
49. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, p. 2. USENIX Association (2012)
50. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. HotCloud **10**, 95 (2010)
51. Zhang, C., Lu, J., Xu, P., Chen, Y.: Unibench: a benchmark for multi-model database management systems. In: Proceedings of the TPCTC (2018)
52. Zhang, K.Z.: Consumer behavior in social commerce: a literature review. Decis. Support Syst. **86**, 95–108 (2016)