Unified Models and Framework for Querying Distributed Data Across Polystores

Léa El Ahdab, Imen Megdiche, André Péninou, Olivier Teste

Unified Models and Framework for querying Distributed Data across Polystores

Léa EL AHDAB[1,4], Imen MEGDICHE[2,4], André PENINOU[3,4], and Olivier TESTE[3,4]

[1] Université de Toulouse, INU Champollion, France
[2] Université Toulouse Jean Jaurès UT2J, France
[3] IRIT, France
{lea.el-ahdab,imen.megdiche,andre.peninou,olivier.teste}@irit.fr

Abstract. Combining data sources from NoSQL and SQL systems leads to data distribution and complexifies user queries: data is distributed among different stores having different data models. This data implementation complexifies the writing of user queries. This work proposes a querying framework of a polystore with the use of unified models as a user vision of the polystore. Unified models hides the variety of data models and data distribution to the user. Our solution uses the Entity-Relationship model of the polystore to infer unified models and to identify intermediate required operations to fulfill querying on real polystore. Using these required transformations, a rewriting framework allows to automatically rewrite the user query (against the unified model) with respect to the real data distribution over the polystore. We apply this framework with one dataset (UniBench benchmark) between a relational, a document-oriented and a graph-oriented databases. We illustrate in this work performance and the low impact of our query rewriting solution when compared to query execution time.

Keywords: Polystore · NoSQL · SQL · Data distribution · Data fragmentation · Query rewriting

1 Introduction

With multi-store and polystore systems [1], the combination of schema systems and schema-less systems have led to distribution and querying issues [2]. One native language is not sufficient to interrogate data. This system heterogeneity emphasizes data complexity for polystore interrogation. Some hybrid languages were proposed in order to query a polystore [3] [4] while others have chosen to adapt existing native languages with developed complementary functions [5]. The main stake is to create a link between different systems in order to have an integrated vision of data. Some works focus on adding an external algorithm to handle heterogeneity of systems [6]. User querying and the vision of the polystore need to be simplified. Logical views or models of each data system translate the physical implementation inside the polystore as if it is only stored in one system

[7]. In this work, we present a framework able to query over heterogeneous polystores with data distribution and fragmentation based on the notion of unified modeling. It shows all data present in the polystore in a single model: relational, document or graph and hides the underlying data distribution, data models, and data fragmentation. This simplified vision is intended to provide transparency and simplicity for user querying. With the generation of a mapping dictionary, our solution is able to create links between the polystore and these unified models. The process may includes the adding of transformation and/or transfer of data. Our main advantage is the independence of our process despite data or structure updates on the polystore. This paper is organized as follows: section 2 presents the main challenges with a motivation example on an e-commerce scenario. Section 3 details the prototype development with the explanation of the framework modules for querying the polystore. Section 4 shows results of the query rewriting and execution time using this framework on real data and over vertical data distribution. Finally we position our work in section 5 and we conclude and give some perspectives about the future ones in section 6.

2 Motivating example

Prerequisite A **polystore** PL is considered as a set of **databases** $\{DB_i\}$. Each database is composed **datasets** DS_j corresponding to the storage of entity classes and their data values. All datasets of one database are from the same family system but it is different for the databases of a polystore: data has different native forms. A *vertical distribution* is the distribution of attributes inside different databases. It can lead to data fragmentation where the distributed attributes belong to the same entity. The information linked to this entity is then fragmented between different datasets. In our context of polystores, it complexifies data distribution with the diversification of native forms. We consider data fragmentation where an entity class is distributed in several databases of the polystore. The entities distributed are linked according to a specific attribute called *distribution key*. This key is the unique value defining an instance of the entity class. It is used to identify all fragments of this entity.

A motivating example. Considering an E-commerce scenario from Unibench, data is distributed into three systems: relational (DB_1,DB_2), document-oriented (DB_3) and graph-oriented (DB_4). This example uses $Customers$, $Products$, $Orders$, $Reviews$, $Persons$, $Post$ and Tag entity classes (Figure 1). For vertical ***data distribution***, the entity class $Customers$ is stored in one dataset DS_1 of one database DB_1 as shown in Figure 1. Data shows ***fragmentation*** where one entity can have its attributes distributed in two databases. For example, the entity class $Products$ is stored in two dataset DS_{2a} of database DB_2 and DS_{2b} of database DB_3. These databases are respectively from a relational storage system and document-oriented system (Figure 1). One key attribute links the two fragments of $Products$ and is called the attribute of fragmentation, $product\_id$. Data fragmentation is illustrated with the vertical fragmentation of $Orders$ and

Products between the relational and the document-oriented system. Their primary key is also the fragmentation key for the distributed entity classes. To hide the complexity of four databases with three different modeling paradigms to the user, we introduce a *unified model*. It shows all data of the polystore "as if" it was a mono-store, thus hiding the real data distribution, data models, and data fragmentation. It is not implemented physically. Figure 2 shows the relational, document-oriented and graph-oriented unified models of the e-scenario data model used for this example.

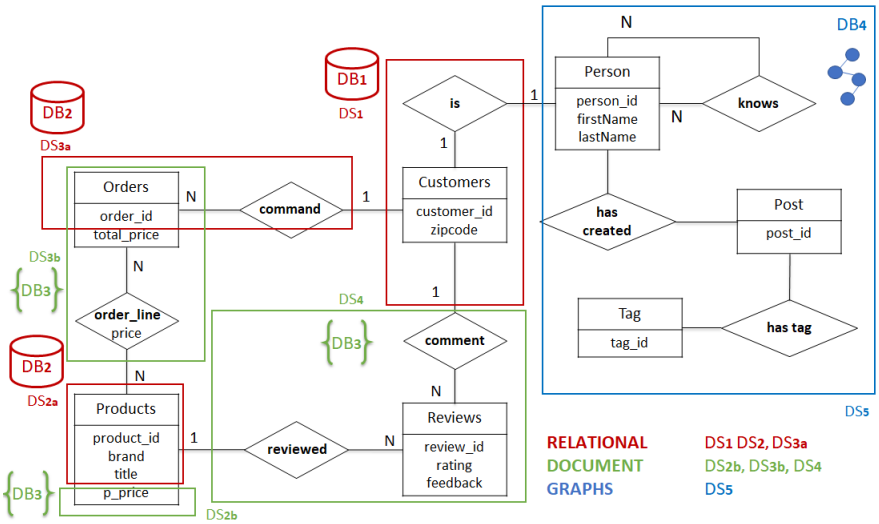

Fig. 1. Data model for the adapted e-commerce scenario

Query use case: "Do customers ordering the same products have a link between each other?" Let's assume that this use case is formulated in SQL on the Relational Logical model. It implies to access to *Person*, *Customers*, *Orders* and *Products* entities stored in different *DB* systems. *Person* is stored in a graph database which is not compatible with the initial query language and system interrogated. Detailing entity location complexifies user query. *Orders* and *Products* are both fragmented between a relational database and a document-oriented database which implies query adaptation and potentially an entity rebuilding. In some works [6, 9] attribute location is specified in the user query. It complexifies the rewriting of the query because the user should be aware of each system, each language, each database and each dataset of the polystore. Figure 3 shows the difference between a user query in CloudMdSQL and a user query on our system.

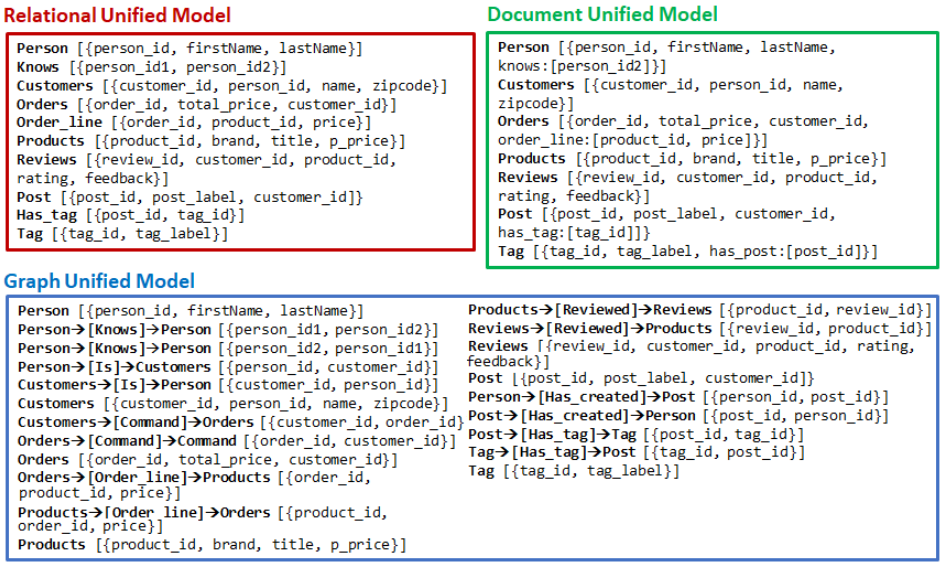

Fig. 2. Unified models of the data model showed in Figure 1

USER QUERY ON CLOUDMDSQL

Customers(customer_id int, person_id int)**@DB1** = (SELECT customer_id FROM Customers)
Products (product_id string)**@DB2** = (SELECT product_id FROM Products)
Orders1(order_id string, customer_id int)**@DB2** = (SELECT order_id, customer_id FROM Orders)
Orders2(order_id string, product_id string)**@DB3** = (db.orders.aggregate({$project:{_id:0, order_id:1, product_id:1}}))
Person (person_id1 int, person_id2 int)**@DB4** = (MATCH (p1:Person)-[k:KNOWS]→(p2:Person) RETURN p1.person_id as person_id1, p2.person_id as person_id2)
SELECT C1.customer_id, C2.customer_id
FROM Customers C1, Customers C2, Person P1, Person P2, Orders1 O11, Orders1 O12, Orders2 O21, Orders2 O22, Products P
WHERE P.product_id=O21.product_id **AND** P.product_id=O22.product_id
AND C1.customer_id = O11.customer_id **AND** C2.customer_id=O12.customer_id
AND O11.order_id=O21.order_id **AND** O12.order_id=O22.order_id
AND (C1.person_id=P1.person_id **OR** C1.person_id=P2.person_id)
AND (C2.person_id=P1.person_id **OR** C2.person_id=P2.person_id);

USER QUERY WITH OUR SYSTEM

SELECT C1.customer_id, C2.customer_id
FROM Customers C1, Customers C2, Knows K, Orders O1, Orders O2, Products P
WHERE P.product_id = O1.product_id **AND** P.product_id=O2.product_id
AND C1.customer_id = O1.customer_id **AND** C2.customer_id = O2.customer_id
AND (C1.person_id = K.person_id1 **OR** C1.person_id = K.person_id2)
AND (C2.person_id = K.person_id1 **OR** C2.person_id = K.person_id1);

Fig. 3. User query comparison between CloudMdsQL and our system for the use case

3 Our proposed framework

3.1 Problem statement

The scope of our contribution considers a polystore PL with databases DB_i belonging to SQL and NoSQL paradigms. Data is distributed between relational, document-oriented and graph-oriented databases. Querying this heterogeneous polystore with a native language L is impossible without L being adapted to all paradigms. This shows two sub-problems: (i) How to provide a global view of PL in one of DBMS type ? (ii) How to execute the query written in an arbitrary chosen language L over PL ?

3.2 An overview of the framework

Our solution aims to provide a transparency querying system over a heterogeneous polystore. A simpler representation of the polystore is used by the user to express his query in one language of his choice. It is analyzed and transformed part by part to take into account the real data location inside PL and to return results in the expected form. Our rewriting system considers data transfer and transformation and favors the use of DBMS operators and performance. Our solution is composed of two phases: the construction phase and the exploitation phase (Figure 4).

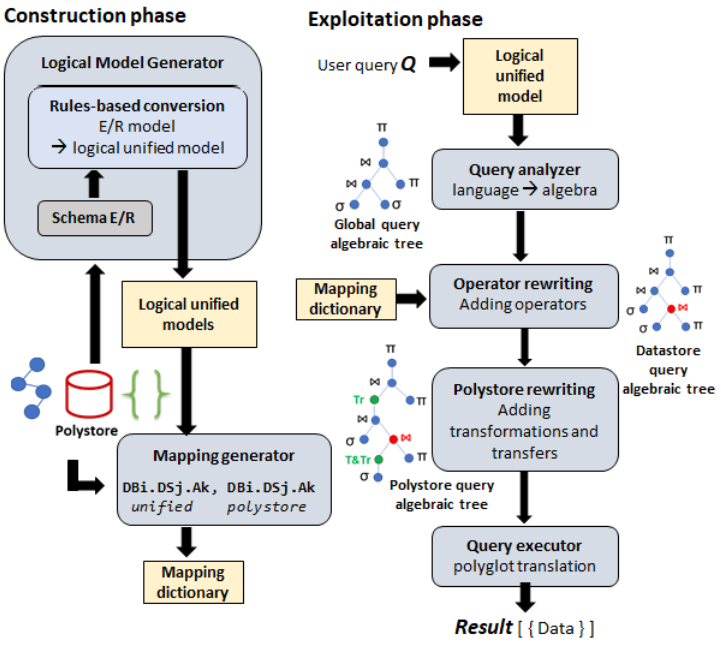

Fig. 4. Overview of our framework for the construction and the exploitation phases

1. **Construction phase:** composed of a module for logical unified model generation and a module for mapping dictionary generation:
   - *Logical Model generator:* gives a unified representation of the polystore for each included system (SQL, NoSQL). It hides the polystore complexity by representing it in a set of databases belonging to the same system (relational, document-oriented or graph-oriented). There is one U_M per system of the polystore. The E/R model of the polystore is the input of this module. It includes all existing attributes of the polystore. It should be given to the module; the automatic extraction of such model is our of the scope of this paper. The U_M generation is based on specific rules: (i) Each entity class becomes a *relation* for the relational unified model, a *collection* for the document unified model and a *category of node* for the graph unified model; (ii) The relationships become two oriented edges linking their respective entity classes for the graph unified model. In case of (1,N) cardinality, the foreign key is integrated into the relationship to N and association properties are new attributes added to this relation for the relational unified model. For the document unified model, the foreign key and relationship attributes become attributes of the collection on the side 1. The relationship corresponds to a nested attribute for the collection on the side N and group the foreign key and the relationship attributes. In case of (N,M) cardinality, the relationship is a nested attribute for both collections grouping the relationship attributes and the respective foreign key for the document unified model. For the relational unified model, the relationship becomes a relation where its primary key is the composition of the primary key of each class of the association.

Algorithm 1 Mapping dictionary generation

```
input: U_V, PL            → unified model, polystore
output: Mapp              → mapping dictionary
Mapp ← {'Type' : U_V.type,' uv_tables' : [ ] }
for DS_i ∈ U_V do
    ds_infos ← {'name' : DS_i.name,' stores_infos : [ ] }
    for pl_db ∈ PL do
        stores ← {'name' : pl_db.name,' type' : pl_db.type}
        stores['columns'] ← [ ]
        for pl_ds ∈ pl_db do
            for attr ∈ pl_ds do
                column_map ← found(attr, PL, UV)
                stores['columns'] ← add(column_map)
            end for
        end for
        ds_infos['stores_infos'] ← stores
    end for
    Mapp['uv_tables'] ← ds_infos
end for
return Mapp
```

– *Mapping generator*: defines links between attributes inside the unified models and their actual location inside the polystore. It is visually represented as a table but it is implemented as a JSON file. The algorithm 1 generates these links according to the unified model and the polystore. For each property of the unified model, its equivalences are found in PL by browsing each dataset of each database of the polystore. Once, the correspondences are found with *found()* function, the result are added to the final mapping structure (with the function *add()*). Table 1 shows an example of the mapping dictionary for the relational unified model. Because the entity classes $Orders$ and $Products$ are fragmented between DB_2 and DB_3 this mapping dictionary allows our framework to link the real position of each attributes of these entity classes with their unified position presented to the user. $Order\_line$ is a relationship with (N,M) cardinality (Figure 1), it is represented as a table in the U_{VR} when applying the *Logical Model generator*. The attributes of $Order\_line$ in the polystore are stored in the $Orders$ table for $DS_{3a}.order\_id$ and in the $Orders$ collection for $DS_{3b}.order\_id$ for the rest of the attributes.

Table 1. Extract of the dictionary showing the entity/relationship between $Orders$ and $Products$ for the relational unified model

U_{VR}	DB_2	DB_3
Orders.order_id	DS_{3a}.order_id	DS_{3b}.order_id
Orders.total_price		DS_{3b}.total_price
Order_line.order_id	DS_{3a}.order_id	DS_{3b}.order_id
Order_line.product_id		DS_{3b}.product_id DS_{3b}.details.product_id
Order_line.price		DS_{3b}.price DS_{3b}.details.price
Products.product_id	DS_{2a}.product_id	DS_{2b}.product_id
Products.brand	DS_{2a}.brand	
Products.title	DS_{2a}.title	
Products.p_price		DS_{2b}.p_price

2. **Exploitation phase:** This phase is composed of a module for query analysis, a module for operator rewriting, a module for optimization and the final module for query execution. For the relational unified model, we consider the operators of selection, projection and join in SQL, for the document unified model, we consider the operators of selection and projection in MongoDB.

   – *Query analyzer:* translates the user query into a global query algebraic tree G_T defined as $G_T = \{N_{node} \rightarrow (N_{left}, N_{right})\}$ where each node N is $N = (op, [E.A])$ and contains information about the associated operation op, the list of attributes accessed by this operation and the

corresponding entity $E.A$. The last nodes are the entities interrogated and has no children nor operations;
- *Operator rewriting:* works on the G_T and specifies data location. The entity classes are changed into their corresponding datasets. In case of fragmentation, we introduce a rebuilding operator ρ to the new algebraic tree. This step adapts the query to the databases of the polystore and generates a multi-store query algebraic tree;
- *Polyglot rewriting:* works on the multi-store query algebraic tree and generates the final polystore query algebraic tree with the presence of transfers and transformations operations. These new nodes appear when there is a change of paradigms identified by the information given in the previous step;
- *Polyglot executor:* generates an execution plan according to the polystore query algebraic tree and translates the sub-trees of one algebraic tree into the respective languages (SQL for relational, MongoDB for document, Cypher for Graphs). This step follows the paradigms transformation rules presented in Table 2. The steps of transformation and transfers of sub-results are included between the execution of those sub-queries. The algebra for the document-oriented and the graph-oriented systems are proposed for the purpose of this work and are not a generalization.

Table 2. Operator equivalences for relational, document-oriented and graph-oriented systems

Operation System	**selection**	**projection**	**join**
Relational algebra - SQL	σ WHERE	π SELECT	$\bowtie$ FROM + WHERE
Document algebra - MongoDB	σ $match	π $project	$\lambda + \mu$ $lookup + $unwind
Graph algebra - Cypher	σ WHERE	π RETURN	$\rightarrow$ MATCH

4 Experiments

In this section, we evaluate with our framework according to two problems: **E1 (Adaptability):** Mapping dictionary generation time, query rewriting time and query execution time when there is a change in the polystore entity classes distribution between systems; **E2 (Volume):** Query execution time for rewritten queries according to volume variation.

4.1 Datasets

Benchmarks in the literature consider data that can be stored in multi-model polystores. *Unibench* is one of them and considers the following systems: key/value,

document-oriented, graph-oriented and relational. In our work, we use the relational, the document-oriented and the graph-oriented models. We extracted data from the University of Helsinki website [8] and we considered data from the JSON (document), the relational, the graph and the key/value systems. For the purpose of experimenting our framework, we decided to convert the key/value data into document-oriented data (because we do not support key/values systems for the moment). We work with vertical data distribution where one entity class is stored in one dataset of one database and on data fragmentation where one entity class is distributed in multiple datasets (in the same or in different databases).

4.2 Developed framework modules

The scope of our theoretical solution considers queries in SQL, MongoDB and Cypher corresponding respectively to relational, document-oriented and graph-oriented systems. We have developed the rewriting module that takes as an input a relational user query (with $\sigma, \pi, \bowtie$ operators) or a document user query (with σ, π operators). It generates its algebraic tree which is then rewritten with new operators. The graph user query is manually analyzed into its polystore algebraic tree. The unified model generation and the execution plan are manually produced. The obtained plan is manually rewritten into a python program to allow its execution over the databases. Their automation is a work perspective.

4.3 Experimental setup and protocol

The experiments were performed on a machine working with Intel i7 2.30GHz and with 64 GB RAM. The polystore data is stored following the distribution scenarios in a version 5.1.1 of MySQL, in MongoDB 5.0.6-rc1 Enterprise of MongoDB Compass and in the version 1.5.9 Neo4J desktop. Our framework is implemented with Python 3.10.2 (jupyter lab) for the construction phase and the creation of the execution plan; and on IDE Netbeans 18 with JAVA 20 and Maven 3.9.2 for the query rewriting phase. We have implemented python files in order to generate relations, collections, type of nodes and type of relationships corresponding to data distribution for the experiments. They use Unibench files to generate new instances to increase the data volume. All new instances are then inserted into the different databases to fulfill the presented experiments. The considered use cases answer the following queries: (i) Q_1 *All products from the brand '54' with a price below 50$*, (ii) Q_2 *Every person that shares a rating of 3 on a product's review*, (iii) Q_3 *All orders for customers coming from the same geographical area*. With our framework, we have tested these queries expressed in SQL, MongoDB and Cypher applied on their respective unified views.

4.4 Evaluation of our framework adaptability

We focus our experiments on our framework ability to adapt according to polystore dataset distribution. dd_1 is the reference distribution, dd_2 is a distribution

without data fragmentation and dd_3 presents fragmentation in two datasets of the same database. These distributions are shown in Table 3. Unified models are based on the E/R model of data stored in the polystore, they are not impacted by the change of dataset distribution inside the polystore. Papping dictionaries depend on properties location inside polystore and they are regenerated following the new implementation. For each distribution, the time for mapping dictionary generation does not exceed 0.01 seconds.

Table 3. Data distribution scenarios for the E1

System DB Distribution	Relational DB_1	Relational DB_2	Document DB_3	Graph DB_4
dd_1	$Customers$	$Products$, $Orders$	$Products$, $Orders$ $Reviews$	$Person$, $Post$ Tag
dd_2	$Customers$	$Products$, $Orders$	$Reviews$	$Person$, $Post$, Tag
dd_3	$Customers$		$Reviews$, $Orders_1$ $Products_1$, $Orders_2$ $Products_2$	$Person$, $Post$ Tag

Each query (Q_1, Q_2, Q_3) is tested using our framework to generate the final algebraic tree according to the polystore configuration. In these scenarios, the initial query was rewritten three times in order to get the average time of algebraic tree rewriting in our exploitation phase. Tables 4 and 5 show the average rewriting time for each dataset distribution and for each query when executed on the relational unified model and on the document unified model. It does not exceed 3 seconds. Depending on the distribution, the operators included can be different for the same query. Q_1 has one join, one transfer and one transformation when rewritten in dd_1 and does not have these operators when rewritten in dd_2 for the relational U_M. In comparison, for the document unified model, the operators of Q_1 in dd_1 are the same but in dd_2 the query has one transfer and one transformation. Q_2 works with two entities that keep the same place in the polystore whatever the considered distribution is, but they are in different systems.

Based on the algebraic trees, execution plans of each query are obtained and their execution time was compared. We focus on Q_1 expressed on the relational unified model and over the three polystore configuration considered. The results returned show that the transfers and transformations from a paradigm to another are the more important factors for execution time increase. Using results of Table 4, the presence of transfer and transformation operators influence the execution time of the initial query. The high number of sub-results that will be converted and transferred in a specific distribution is the reason why the execution time is higher than for another distribution scenario.

Table 4. Results of algebraic tree generation for each query for each data distribution inside the polystore for the SQL interrogation of the relational unified model

Query Distribution	Q_1 dd_1	Q_1 dd_2	Q_1 dd_3	Q_2 dd_1	Q_2 dd_2	Q_2 dd_3	Q_3 dd_1	Q_3 dd_2	Q_3 dd_3
Average rewriting time (seconds)	**0.76**	**0.86**	**0.73**	**2.36**	**2.28**	**2.32**	**0.88**	**0.86**	**0.87**
Number of joins	1	0	1	2	2	2	2	1	2
Number of transfers	1	0	1	2	2	2	2	1	2
Number of transformations	1	0	1	2	2	2	1	0	2

Table 5. Results of algebraic tree generation for each query for each data distribution inside the polystore for the MongoDB interrogation of the document unified model

Query Distribution	Q_1 dd_1	Q_1 dd_2	Q_1 dd_3	Q_2 dd_1	Q_2 dd_2	Q_2 dd_3	Q_3 dd_1	Q_3 dd_2	Q_3 dd_3
Average rewriting time (seconds)	**1.28**	**1.46**	**1.32**	**1.58**	**1.55**	**1.82**	**1.54**	**1.57**	**1.62**
Number of joins	1	0	1	2	2	2	1	1	1
Number of transfers	1	1	0	2	2	2	2	2	1
Number of transformations	1	1	0	2	2	2	1	1	1

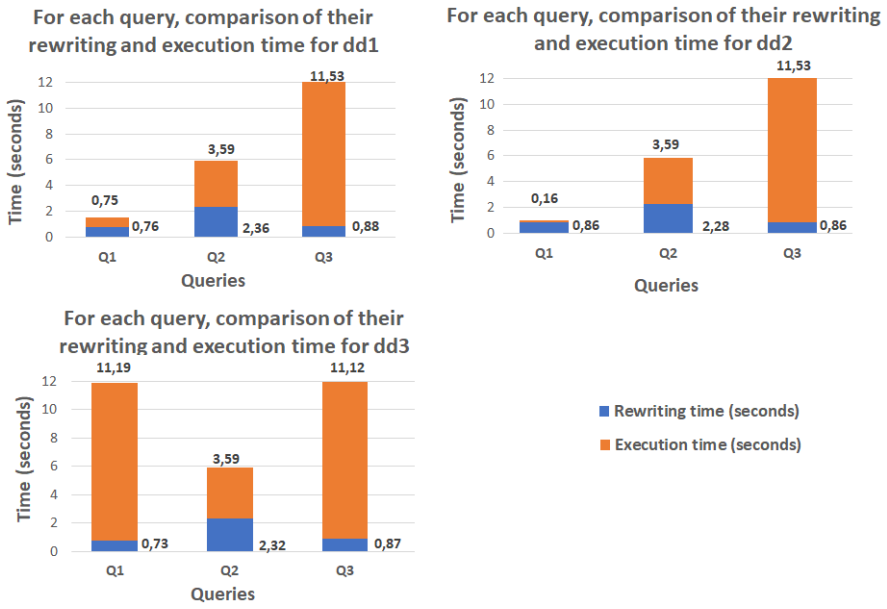

Fig. 5. Experiment result for query rewriting time and query execution time for the three polystore distribution considered in this paper considering the execution over the relational unified model

Figure 5 shows execution results and rewriting results for each query on each distribution inside the polystore. Q_2 presents the maximum of transfers and transformation needed to address data inside the polystore properly, whereas Q_1 only presents one transfer and one transformation and Q_3 is composed of two operations of transfer and only one of transformation. For Q_1, the change of distribution for the polystore impacts its execution time when the entity class is fragmented between two paradigms (0.75 s), is not fragmented (0.16 s), and fragmented in a different paradigm than the one interrogated (11.19 s). The two entity classes considered by Q_2 are in two different paradigms. One of the sub-results needs to be transformed and transferred into the relational one, which explains the execution time (3.59 s). Q_3 execution time (11.53 s) is not impacted by the change of dd_1 with dd_2. In these scenarios, the entity classes considered belong to the same paradigm even if they are found in two databases. In dd_3, one entity class changes its paradigm and needs a transformation operation before the transfer one. The execution time is still close to the one for dd_1 and dd_2 (11.12 s).

4.5 Evaluation of our framework with data volume

This experiment is based on the dd_1 dataset distribution where we added more instances to double the initial data volume as shown in Table 6.

Table 6. Data volumes and detail of instances for each DB_i of PL

Volume	DB_1	DB_2	DB_3	DB_4
V_1 (**103.09** Mb)	0.46 Mb	12.4 Mb	3.23 Mb	87 Mb
V_2 (**200** Mb)	15 Mb	55 Mb	15 Mb	115 Mb

New instances in V_2 are randomly generated from the initial Unibench dataset using a python algorithm and dependencies between entities are respected. We experiment how data volume impacts query execution. These modifications are linked to the number of instances, logical unified models and the mapping dictionary are not impacted. The rewriting time of each query is the same than in the previous experiment for dd_1 however the execution time is different. Q_1 shows a rewriting variation and the location of all attributes of *Products* is searched using the mapping dictionary. Depending on the original system chosen, the transfer and transformation operations are added to the algebraic tree. The increase of volume is expected to impact the number of lines returned for each sub-result and hence, the execution time. Q_1 is composed of two sub-queries: q_1 corresponding to the first selection on the *brand* attribute and q_2 corresponding to the second selection on the *price* attribute. The results obtained for Q_1 execution time comparing V_1 and V_2 are presented in Figure 6. This experiment is based on an execution plan manually generated for each unified model

interrogated. We expected an execution time twice higher for V_2 compared to V_1. We obtained an increase of execution time for V_2 that is almost 1,5 times higher than V_1. Q_1 considers *Products* entity which is fragmented between the relational database and the document-oriented database as considered in dd_1 data distribution. The execution time for the document-oriented unified model is explained by the low number of lines transformed and transferred from the relational database into the document database when compared to the execution time for the relational-oriented database which needs the transformation and transfer of thousands of lines. Finally for the graph oriented model, there are more transfers and transformations from the relational and document databases which explain the high value in Figure 6.

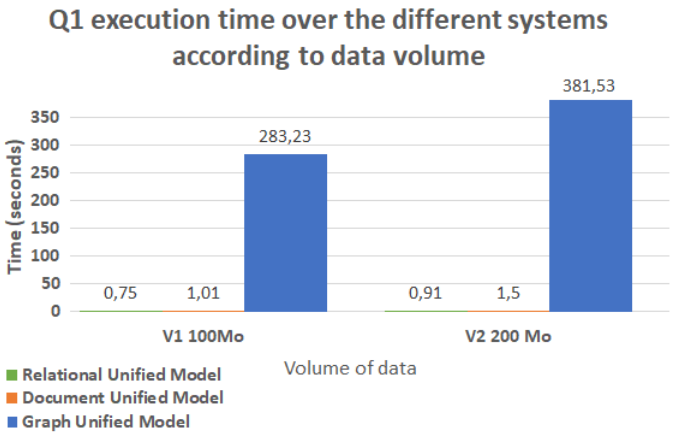

Fig. 6. Experiment result for query execution time for the two data volumes V_1 and V_2 considering the execution over the relational, document-oriented and graph oriented unified models for Q_1

5 Related Work

The appearance of polystores with the combination of SQL and NoSQL systems leads to different level of complexity: data distribution, different models and query language.

Data distribution. The association of several systems leads to data distribution. Existing works focus on vertical distribution where one entity class is found in one dataset of one database of the polystore. Accessing data of each system is complicated when crossing systems. Operations are executed outside DBMS with an external function [6] [14]. HydRa [5], a framework proposing physical possible models according to a specific conceptual model, mentions this type of

vertical distribution for one entity but do not explain how to considerate it for querying purposes. Some solution [6] might apply to data fragmentation. Their approach uses an algorithm executing the join operation between datasets of different databases, we suppose they can handle this specific case of vertical data distribution but they do not experiment this in their work.

Data Model. Some works try to find a universal representation for a polystore to hide the complexity into one model. This schema inference helps to define the multiple systems as a simplified one [10]. It can be a graph representation [11] [12] [13] or a u-schema model [14] illustrating structural variations. Because "one size does not fit all" [15], it is question to transfer all data into this new model. Changing data representation impacts users and modifies the initial paradigm presented to them. They must adapt to one fixed vision of the polystore.

Query language. Having different data storage systems brings the question of the user query language. Some works choose to rewrite the initial query according to user specifications or to one system of the polystore [16] [17]. Some works [18] use parallel query methods outside data stores (map/reduce/filter). It is helpful to execute operators outside DBMS and to not consider paradigms conversion and they use specific languages limited to relational operators (CloudMdsQL language [6]). BigDAWG [9] uses the principle of islands which communicates with adapters. To query this polystore, the user needs to specifies the system interrogated (for example $RELATIONAL(\ SELECT\ *\ FROM...)$). This adds information to the user query. In this paper, we choose to provide unified models to allow the user to query with transparency because our systems deals with the sub-queries generated from the mono-language user query.

Overview. Table 7 illustrates the differences between our works and others working on vertical data distribution inside polystores. The comparison is for the relational R, document-oriented D, column-oriented C and graph G systems, the query language(s) considered and if it is question of entity class distribution in one or several system (fragmentation). In our work, the user has the choice of the unified model he wants to query without thinking about the distribution and possible fragmentation inside the polystore interrogated.

6 Conclusion

In this paper, we focus on polystore systems with relational, document-oriented and graph-oriented systems. Data is distributed vertically. We define a framework composed of unified models intended to hide the polystore complexity to users. A mapping dictionary is generated to link these representations and the real data distribution. The user can query with a single query one unified model of his choice (relational, document, graph) whereas data is kept in its native form and in its specific location inside the polystore. Experiments were conducted on a Unibench dataset, showing the low impact of data distribution on

Table 7. A comparison of existing solutions on polystores

Authors	R	D	C	G	Query	Entity class fragmentation
El Ahdab et al	●	●	○	●	*SQL - MongoDB - Cypher*	●
Barret et al [12]	●	●	○	●	SparkQL	○
Candel et al [14]	●	●	○	●	SQL	○
Ben Hamadou et al [13]	●	●	●	○	SQL - MongoDB	○
Hai et al [16]	●	●	○	●	SQL - JSONiq	○
Papakonstantinou [17]	●	●	○	○	SQL	○
Duggan et al [9]	●	●	●	○	Declarative	○

the rewriting solution and the more important impact on the execution time. Unified models depend only of data structures, they need to be rebuilt only when some data structures change in the polystore (documents or graphs). Existing works on data model extraction may help to automatically produce E/R model. The execution plan optimization and automation will be a prospect of development. This part has an important cost due to data transfers and transformation. Considering our future work on polystore systems, we will expend our operators for each considered model (aggregation), data models (key-value) and we will experiment graph queries on graph unified model.

Acknowledgments. This work was supported by the French Government in the framework of the Territoire d'Innovation program, an action of the *Grand Plan d'Investissement* backed by France 2030, Toulouse Métropole and the GIS neOCampus.

References

1. LECLERCQ, Éric et SAVONNET, Marinette. A tensor based data model for polystore: an application to social networks data. In : Proceedings of the 22nd International Database Engineering & Applications Symposium. 2018. p. 110-118.
2. FORRESI, Chiara, GALLINUCCI, Enrico, GOLFARELLI, Matteo, et al. A dataspace-based framework for OLAP analyses in a high-variety multistore. The VLDB Journal, 2021, vol. 30, no 6, p. 1017-1040.
3. RAMADHAN, Hani, INDIKAWATI, Fitri Indra, KWON, Joonho, et al. MusQ: a Multi-store query system for iot data using a datalog-like language. IEEE Access, 2020, vol. 8, p. 58032-58056.
4. MISARGOPOULOS, Antonis, PAPAVASSILIOU, George, GIZELIS, Christos A., et al. TYPHON: Hybrid Data Lakes for Real-Time Big Data Analytics–An Evaluation Framework in the Telecom Industry. In : IFIP International Conference on Artificial Intelligence Applications and Innovations. Cham : Springer International Publishing, 2021. p. 128-137.
5. GOBERT, Maxime, MEURICE, Loup, and CLEVE, Anthony. HyDRa A Framework for Modeling, Manipulating and Evolving Hybrid Polystores. IEEE Interna-

tional Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2022. p. 652-656.
6. KOLEV, Boyan, VALDURIEZ, Patrick, BONDIOMBOUY, Carlyna, et al. CloudMdsQL: querying heterogeneous cloud data stores with a common language. Distributed and parallel databases, 2016, vol. 34, p. 463-503.
7. EL AHDAB, Léa, TESTE, Olivier, MEGDICHE, Imen, et al. Unified views for querying heterogeneous multi-model polystores. In : International Conference on Big Data Analytics and Knowledge Discovery. Cham : Springer Nature Switzerland, 2023. p. 319-324.
8. Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. 2018. UniBench: A Benchmark for Multi-model Database Management Systems. Proceedings of the Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2018). Rio de Janeiro, Brazil, 7–23.
9. DUGGAN, Jennie, ELMORE, Aaron J., STONEBRAKER, Michael, et al. The bigdawg polystore system. ACM Sigmod Record, 2015, vol. 44, no 2, p. 11-16.
10. KOUPIL, Pavel, HRICKO, Sebastián, et HOLUBOVÁ, Irena. Schema inference for multi-model data. In : Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems. 2022. p. 13-23.
11. BARRET, Nelly, MANOLESCU, Ioana, et UPADHYAY, Prajna. Computing generic abstractions from application datasets. In : EDBT. 2024.
12. BARRET, Nelly, MANOLESCU, Ioana, et UPADHYAY, Prajna. Abstra: Toward Generic Abstractions for Data of Any Model. 31st ACM International Conference on Information & Knowledge Management. 2022. p. 4803-4807.
13. BEN HAMADOU, Hamdi, GALLINUCCI, Enrico, et GOLFARELLI, Matteo. Answering GPSJ queries in a polystore: A dataspace-based approach. 38th International Conference, ER 2019, Salvador, Brazil, November 4–7, 2019, Proceedings 38. Springer International Publishing, 2019. p. 189-203.
14. CANDEL, Carlos J. Fernández, RUIZ, Diego Sevilla, et GARCÍA-MOLINA, Jesús J. A unified metamodel for NoSQL and relational databases. Information Systems, 2022, vol. 104, p. 101898.
15. KHAN, Yasar, ZIMMERMANN, Antoine, JHA, Alokkumar, et al. One size does not fit all: querying web polystores. Ieee Access, 2019, vol. 7, p. 9598-9617.
16. HAI, Rihan, QUIX, Christoph, et ZHOU, Chen. Query rewriting for heterogeneous data lakes. Advances in Databases and Information Systems: 22nd European Conference, ADBIS 2018, Budapest, Hungary, September 2–5, 2018, Proceedings 22. Springer International Publishing, 2018. p. 35-49.
17. PAPAKONSTANTINOU, Yannis. Polystore Query Rewriting: The Challenges of Variety. EDBT/ICDT Workshops. 2016.
18. KRANAS, Pavlos, KOLEV, Boyan, LEVCHENKO, Oleksandra, et al. Parallel query processing in a polystore. Distributed and Parallel Databases, 2021, p. 1-39.