ESTOCADA: Towards Scalable Polystore Systems

R. Alotaibi[‡] B. Cautis[§] A. Deutsch[‡] M. Latrache[†] I. Manolescu[†]
Y. Yang [‡]

[†]Inria & Institut Polytechnique de Paris [‡]UC San Diego [§]Univ. Paris-Saclay
{ralotaib, deutsch, yiy032}@eng.ucsd.edu, bogdan.cautis@u-psud.fr,
{moustafa.latrache, ioana.manolescu}@inria.fr

ABSTRACT

Big data applications increasingly involve diverse datasets, conforming to different data models. Such datasets are routinely hosted in heterogeneous stores, each capable of handling one or a few data models, and each efficient for some, but not all, kinds of data processing. Systems capable of exploiting disparate data in this fashion are usually termed *polystores*. A current limitation of polystores is that applications are written taking into account which part of the data is stored in which store and how. This fails to take advantage of (*i*) possible redundancy, when the same data may be accessible (with different performance) from distinct data stores; (*ii*) previous query results (in the style of materialized views), which may be available in the stores.

We propose to demonstrate ESTOCADA [4], a novel approach that can be used in a polystore setting to transparently enable each query to benefit from the best combination of stored data and available processing capabilities. The system leverages recent advances in the area of view-based query rewriting under constraints, which we use to describe the various data models and stored data.

PVLDB Reference Format:
R. Alotaibi, B. Cautis, A. Deutsch, M. Latrache, I. Manolescu, Y. Yang. ESTOCADA: Towards Scalable Polystore Systems. *PVLDB*, 13(12): 2949 - 2952, 2020.
DOI: https://doi.org/10.14778/3415478.3415516

1. INTRODUCTION

Big Data applications increasingly involve *diverse* data sources, such as flat or nested relations, structured or unstructured documents, data graphs, etc. Such datasets are routinely hosted in heterogeneous stores. One reason is that the fast pace of application development prevents consolidating all the sources into a single data format and loading them into a single store. Instead, the data model often dictates the choice of the store, e.g., JSON documents get loaded in a JSON store. Systems capable of exploiting diverse data in this fashion are termed *polystores* [8, 3].

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415516

Query evaluation in a polystore recalls to some extent mediator systems; in both cases, sub-queries are delegated to the underlying stores when possible, while the remaining operations are applied in the integration layer. *Polystores process a query assuming that each of its input datasets is available in one store (chosen for its support of a dataset data model).*

We identify two limitations of such architectures. First, they do not exploit possible data redundancy: the same data could be stored in several stores, some of which may support a query operation much more efficiently than others. Second, they are unable to take advantage of the presence of previously computed query results, which may be available in one or several stores (in the style of materialized views), when the data model of the queried dataset differs from the data model of the store hosting the view.

To overcome these limitations, we have described in [4] ESTOCADA, a novel approach for *allowing an application to transparently exploit data stored in a set of heterogeneous stores, as a set of (potentially overlapping) data fragments;* if fragments store results of partial computations applied on the data, we show *how to speed up queries using them as materialized views, which reduces query processing effort and seeks to take maximum advantage of the efficient query processing features of each store.* Our approach *does not require any change to the application code.* The example below illustrates these performance-enhancing opportunities.

1.1 Motivating Scenario

Dataset. The Medical Information Mart for Intensive Care III (MIMIC-III) [11] dataset comprises health data for more than 40000 Intensive Care Unit (ICU) patients. The total size is 46.6 GB, and it consists of: (*i*) all charted data for all patients and their hospital admission information, ICU stays, laboratory measurements, caregivers' notes, and prescriptions; (*ii*) the role of caregivers and (*iii*) diagnosis related groups' codes descriptions.

Our motivation query Q_1 is: "for the patients transferred into the ICU due to "coronary artery" issues, with abnormal blood test results, find the date/time of their admission, previous location (e.g., clinic referral), and the drugs of type "additive" prescribed to them". Evaluating this query through AsterixDB [1] (v0.9.4) JSON store took more than 25 minutes; this is because the system does not support full-text search by an index if the text occurs within a JSON array. In SparkSQL (v2.3.2), the query took more than an hour, due to its lack of indexes for selective data access. In the MongoDB JSON store (v4.0.2), it took more than 17 minutes due to its limited join capability. Finally, in PostgreSQL with JSON support (v9.6), Q_1 took 12.6 minutes.

Now, consider we had at our disposal three *materialized views*, which pre-compute partial results for Q_1. Consider a SOLR (well-known full-text search engine) sever stores a view V_1 storing the IDs of patients and the caregivers' notes on the patients' stay. Full-text search on V_1 for "coronary artery" allows to efficiently retrieve the respective patients' IDs. Further, consider that a PostgreSQL server stores a view V_2 with the patients information and their hospital admission information such as patients' location prior to admission. Finally, assume available a view V_3, which stores all drugs prescribed for each patient with "abnormal blood" test results, as a JSON document stored in PostgreSQL.

Using these views, we can evaluate Q_1 by a full-text search on V_1 followed by a BindJoin with the result of filtering V_3, and projecting prescribed drugs as well as patients' admission time and prior location from V_2. Using a Polystore Java-based execution engine [4] (implementing select, project, join, etc.) to access the views and join them, this takes about **speedup of 5**× w.r.t. plain JSON query evaluation in SparkSQL and AsterixDB. This is also a **speedup of 2**× and **speedup of 3**× w.r.t. plain JSON query evaluation in MongoDB and Postgres, respectively.

Lessons learned. We can draw the following conclusions from the above example. **(1.)** Unsurprisingly, materialized views improve query performance since they pre-compute partial query results. **(2.)** More interestingly: materialized views can strongly improve performance even when stored across several data stores, although such a hybrid scenario incurs a certain performance penalty due to the marshaling of data from one data model/store to another. In fact, exploiting the different strengths of each system (e.g., SOLR's text search capability) is the second reason (together with materialized view usage) for the performance gains. **(3.)** Different system combinations work best for different queries. Thus it must be easy to add/remove a view in one system, without disrupting other queries that may be currently well-served. As known from classical data integration research [9], such flexibility is attained through the "local-as-view" (LAV) approach, where the content of each data source is described as a view. Thus, adding or removing a data source from the system is easily implemented by adding or removing the respective view definition. **(4.)** Application data sets may come in a variety of formats. However, while the storage model may change as data migrates, applications should not be disrupted. A simple way of achieving this is to guarantee them access to the data in its native format, regardless of where it is stored.

Observe that the combination of **2.**, **3.** and **4.** above goes well beyond the state of the art. Most LAV systems assume both the application data and the views are organized according to the same data model. Thus, their view-based query rewriting algorithms are designed specifically within the bounds of that model, e.g., relational or XML. Different from these, some LAV systems [12, 7] allow different data models for the stored views, but consider only the case when the application data model is XML. As a consequence, the query answering approach adopted in these systems is tailored toward the XML data model and query language. In contrast, we aim at a unified approach, supporting *any data model both at the application and at the view level*.

The core technical question to be answered in order to attain such performance benefits without disrupting applications is *view-based query rewriting across an arbitrary set of data models*. Our research paper [4] covers the technical details of ESTOCADA; here, we overview the system and detail our demonstration walkthrough. Table 1 depicts the supported systems, languages and data models; those added since the publication of [4] appear in bold.

Table 1: Supported Languages and Data Stores

Data model	Query language/API	Systems
Relational	SQL	Major vendors
JSON	SparkSQL	Apache Spark
JSON	AQL/SQL++	Apache AsterixDB
JSON	SQLw/JSON	PostgreSQL
JSON	**MongoDB QL**	**MongoDB**
Array/Tensor	**DML**	**Apache SystemML**
Array/Tensor	**TensorFlow APIs**	**TensorFlow**
Key-value	Redis API	Redis
Full-text and JSON	Solr API	Apache Solr
Graph	**Cypher**	**Neo4j**
XML	XQuery, XPath	Saxon

Relation with prior work. A much earlier version of ESTOCADA was presented in [6]. This demonstration improves over it through several contributions at the level of: (*i*) *modeling*: support for the array (matrix), JSON, and graph models; (*ii*) *algorithms*: optimizing the state-of-the-art algorithm for relational view-based rewriting under constraints to make it scale to a polytsore setting [4], and (*iii*) *systems*, by deploying ESTOCADA on top of three existing polystore systems: BigDAWG [8], SparkSQL, and Tatooine [5] and showing the performance benefits that our approach can bring to these systems.

2. REWRITING PROBLEM STATEMENT

We assume a set of data model-query language pairs $\mathcal{P} = \{(M_1, L_1), (M_2, L_2), \ldots\}$ such that for each $1 \leq i$, an L_i query evaluated against an M_i instance returns an answer which is also an M_i instance. The same model may be associated to several query languages (e.g., AsterixDB and MongoDB have different query languages for JSON).

We consider *expressive languages for realistic settings, supporting conjunctive querying, nesting, aggregation, and object creation*. Without loss of generality, we consider that a language is paired with one data model only.

We consider a polystore setting comprising a set of stores $\mathcal{S} = \{S_1, S_2, \ldots\}$ such that each store $S \in \mathcal{S}$ is characterized by a pair $(M_S, L_S) \in \mathcal{P}$, indicating that S stores instances of the model M_S and evaluates queries expressed in L_S.

We consider a set of datasets $\mathcal{D} = \{D_1, D_2, \ldots\}$, such that each dataset $D \in \mathcal{D}$ is an instance of a data model M_D. A dataset D is stored as a set of (potentially overlapping) **materialized views** $\{V_D^1, V_D^2 \ldots\}$, such that for every $1 \leq j$, V_D^j is stored within the storage system $S_D^j \in \mathcal{S}$. Thus, V_D^j is an instance of a data model supported by S_D^j [1].

We consider a *Polystore* integration language $\mathcal{IL}$ [8, 3], capable of expressing computations to be made within each store and across all of them, as follows:

- For every store S and query $q_S \in L_S$, $\mathcal{IL}$ allows specifying that q_S should be evaluated over S;
- Further, $\mathcal{IL}$ allows expressing powerful processing over the results of one or several such source queries.

[1]For uniformity, we describe any collection of stored data as a view, e.g., a table stored in an RDBMS is an (identity) view over itself.

```
View V1:
FOR AJ:{SELECT  M.patientID AS patientID,
                A.admissionID AS admissionID,
                A.report AS report
         FROM  MIMIC M, M.admissions A}
RETURN SJ:{"patientID":patientID,
           "admissionID":admissionID,
           "report":report}
```

Figure 1: QBT^{XM} Definition of View V_1

We assume available a *cost model* which, given an $\mathcal{IL}$ expression e, returns an estimation of the cost of evaluating e (including the cost of its source sub-queries). We outline the concrete model we use in [4]. We term a *rewriting* of a query q an integration query expressed in $\mathcal{IL}$, which is equivalent to q. We consider the following rewriting problem:

Definition 1 (Cross-model rewriting problem). *Given a query $q \in \mathcal{IL}$ over several datasets D_i, $1 \leq i \leq n$, and a set of views $\mathcal{V}$ materialized over these datasets, find the equivalent rewriting r of q using the views, which minimizes the cost estimation $c(r)$.*

3. ESTOCADA OVERVIEW

We describe below our approach for solving the above rewriting problem and the technical challenges this raises. We refer readers to our paper [4] for details.

Polystore Integration Language. We designed $QBT^{XM} \in \mathcal{IL}$, a concrete polystore integration language. QBT^{XM} follows a *block*-based design, with blocks organized into a tree, each block can be expressed in a different query language and carry over data of a different data model. Other existing polystore integration languages, e.g., [8] resemble QBT^{XM} .

Cross-Model Views. Each materialized view V is defined by an $\mathcal{IL}$ query; it may draw data from one or several data sources, of the same or different data models. Each view returns (holds) data following one data model, and is stored in a data store supporting that model. Figure 1 depicts the QBT^{XM} definition of view V_1 from Section 1. FOR clauses bind variables, while RETURN clauses specify how to construct new data based on the variable bindings. Blocks are delimited by braces annotated by the language whose syntax they conform to. The annotations AJ and SJ stand for AsterixDB's SQL++ and SOLR search, respectively.

Encoding into a Single *Pivot* Model. We reduce the cross-model rewriting problem to a single model setting, namely *relational constraint-based query reformulation*, as follows (see Figure 2; yellow background identifies the areas where we bring contributions in this work.). First, we *encode relationally* the structure of original datasets, the views' definitions and the application query. Note that the relations used in the encoding are *virtual*, i.e., no data is migrated into them; they are also *hidden*, i.e., invisible to both the application designers and to users. They only serve to support query rewriting via relational techniques.

The virtual relations are accompanied by *integrity constraints* that reflect the features of the underlying data models (for each model M, a set $enc(M)$ of constraints). The pivot model is fully detailed in [4].

From Cross Model to Single Model Rewriting. Our reduction translates the declaration of each view V to additional constraints $enc(V)$ that reflect the correspondence between V's input data and its output. Constraints have been

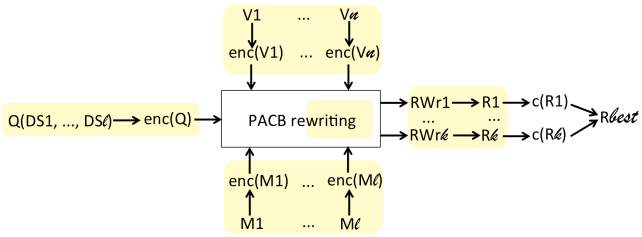


Figure 2: Outline of our approach

used to encode single-model views and correspondences between source and target schemas in data exchange [9]. The novelty here is the rich collection of supported models, and the cross-model character of the views. An incoming query $Q \in \mathcal{IL}$ is encoded as a relational query $enc(Q)$ over the relational encoding of the datasets it refers to.

The reformulation problem is thus reduced to a purely relational setting: given a relational query $enc(Q)$, a set of relational integrity constraints encoding the views, $enc(V_1) \cup \ldots \cup enc(V_n)$, and the set of relational constraints obtained by encoding the data models $M_1, \ldots, M_l$, find the queries RW_r^i expressed over the relational views, for some integer k and $1 \leq i \leq k$, such that each RW_r^i is equivalent to $enc(Q)$ under these constraints.

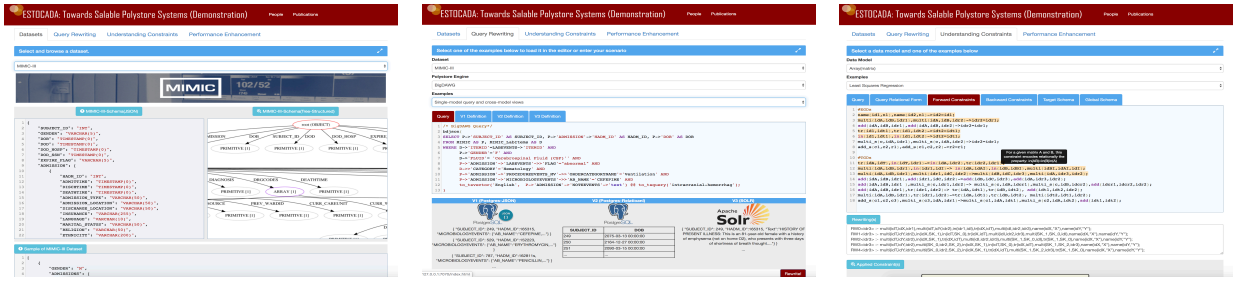
The **challenge** in coming up with the reduction consists in designing a *faithful* encoding, i.e., one in which rewritings found by (i) encoding relationally, (ii) solving the resulting relational reformulation problem, and (iii) decoding each reformulation RW_r^i into a $\mathcal{IL}$ query $R = dec(RW_r^i)$ over the views in the polystore, correspond to rewritings found by solving the original problem. That is, R is a rewriting of Q given the views $\{V_1, \ldots, V_n\}$ if $R = dec(RW_r^i)$, where RW_r^i is a relational reformulation of $enc(Q)$ under the constraints obtained from encoding $V_1, \ldots, V_n$, $M_1, \ldots, M_l$.

Relational Rewriting Using Constraints. To solve the single-model rewriting problem, we need to reformulate relational queries under constraints. The algorithm of choice is the recent *Provenance-Aware* Chase& Backchase algorithm (PACB, in short) [10]. PACB was designed to work with relatively few views and constraints. However, in the polystore setting, each of the many datasources is described by a view, and each view is encoded by many constraints. ESTOCADA also includes a set of optimization techniques we brought to PACB to make it scale in a polystore setting [4].

Decoding Relational Rewritings. On each relational reformulation RW_r^i issued by our modified PACB rewriting engine, a *decoding step* $dec(RW_r^i)$ is performed to translate RW_r into $\mathcal{IL}$ syntax by (i) grouping the reformulation atoms by the view they pertain to, (ii) translating each such atom group into a query which can be evaluated over a single view, and (iii) if several views reside in the same store, identify the largest subquery that can be delegated to that store.

Choice of an Efficient Rewriting. Decoding may lead to several rewritings $R_1, \ldots, R_k$; for each R_i, several evaluation plans may lead to different performance. For each rewriting R_i, we denote by $c(R_i)$ the lowest cost of an evaluation plan that we can find for R_i; we choose the rewriting R_{best} that minimizes this cost. We architectured ESTOCADA to take the cost model as a configuration parameter[2].

[2]Devising cost models for polystore settings is beyond the scope of this paper



(a) Browsing through datasets (b) Query rewriting (c) Understanding constraints

Figure 3: Screenshots of various ESTOCADA demonstration GUIs

4. DEMO WALKTHROUGH

The ESTOCADA approach can be combined with (applied on top of) any polystore, to enhance its performance through exploiting materialized views. We propose to demonstrate it on three existing polystores: BigDAWG [8], SparkSQL, and Tatooine [5], with underlying data stores: PostgreSQL, PostgresSQL with JSON support, Apache Solr, and Apache SystemML. We extended ESTOCADA to support array(matrix) and graph data models in addition to the supported data models in [4].

We propose to show ESTOCADA in action on a set of scenarios similar to the one described in Section 1. We use three datasets: (*i*) MIMIC-III [11], described in Section 1.(*ii*) GDELT [2], which captures worldwide events (e.g., protests, peace appeals) and consists of the events' structural information (e.g., event's id and date), events' actors, geography, and various links to articles describing each event. The dataset is in a relational form, and 26GB (2015 events) in file size. (*iii*) GENOMICS [13], comprising patients, gene metadata and microarray data (matrix form-dense). The size of the dataset is 32GB.

Demo attendees can interact with the systems through the following interfaces:

Datasets Interface. This allows to browse, for each data source, the schema (or its structural summary, for semistructured data such as JSON lacking a schema), and sample of the data, to get familiar with its structure and help formulate queries (see Figure 3(a)).

Query Rewriting Interface. The user can interact with this interface by selecting a dataset, a polystore system query language (QBT^{XM}, SparkSQL or BigDAWG) to use, and a query (expressed in the chosen polystore's system query language) from a workload we prepared, as shown in Figure 3(b). Each dataset comes with a previously specified workload, and a set of stored overlapping/redundant views (their definitions, storage model, and location will be shown to the user); the users also have the option to write their queries and views. On the same interface, the user can inspect the cross-model query rewriting, its translation in the pivot model, and its translated form in the chosen language.

Understanding Constraints Interface. This interface enables the users to select one of the supported data models, inspect the relational integrity constraints that capture the properties of that model, disable/enable/add some constraints, and observe the query rewritings thus obtained. In the same interface, we provide a real-time visualization of which constraints are exploited by the rewriting engine.

Performance Enhancement Interface. In this interface, users can see the performance comparison between a native polystore query execution and the one enabled by cross-stores views-based rewriting.

5. CONCLUSION

The goal of our demonstration is to show that query evaluation performance in polystore architectures can be significantly enhanced by exploiting materialized views. In such a setting, operating over multiple data stores, with potentially different data models, dramatic performance enhancements can be attained by materializing the right view in the right store, that is: the one which can provide the best performance for the operations that need to be applied on it. ESTOCADA supports this functionality by enabling local-as-view query rewriting over a large range of data models and heterogeneous stores. Its immediate benefit is flexibility since it enables taking advantage in an optimal way of any combination of underlying data stores. At the technical core of ESTOCADA lies the constraint-based query rewriting.

As part of our future work, we will devise cost-based recommendation algorithms to pick the best possible placement of a materialized view.

6. REFERENCES

[1] AsterixDB. https://asterixdb.apache.org/.

[2] GDELT.https://www.gdeltproject.org/data.html.

[3] D. Agrawal et al. RHEEM: enabling cross-platform data processing - may the big data be with you! *PVLDB*, 11(11):1414–1427, 2018.

[4] R. Alotaibi et al. Towards scalable hybrid stores: Constraint-based rewriting to the rescue. In *SIGMOD*, 2019.

[5] R. Bonaque et al. Mixed-instance querying: a lightweight integration architecture for data journalism. *PVLDB*, 9(13):1513–1516, 2016.

[6] F. Bugiotti et al. Flexible hybrid stores: Constraint-based rewriting to the rescue. In *ICDE*, 2016.

[7] A. Deutsch et al. MARS: A system for publishing XML from mixed and redundant storage. In *Proc. of VLDB*, pages 201–212, 2003.

[8] J. Duggan et al. The BigDAWG polystore system. In *SIGMOD*, 2015.

[9] A. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[10] I. Ileana et al. Complete yet practical search for minimal query reformulations under constraints. In *SIGMOD*, 2014.

[11] A. Johnson et al. MIMIC-III. *Available at: http://www.nature.com/articles/sdata201635*, 2016.

[12] I. Manolescu et al. Answering XML queries on heterogeneous data sources. In *Proc. of VLDB*, pages 241–250, 2001.

[13] R. Taft et al. Genbase: A complex analytics genomics benchmark. In *SIGMOD*, 2014.