# Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue

Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, Stamatis Zampetakis

# Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue

Rana Alotaibi
UC San Diego
ralotaib@eng.ucsd.edu

Damian Bursztyn*
Thales Digital Factory,France
dbursztyn@gmail.com

Alin Deutsch
UC San Diego
deutsch@cs.ucsd.edu

Ioana Manolescu
Inria
LIX (UMR 7161, CNRS and
Ecole polytechnique)
France
ioana.manolescu@inria.fr

Stamatis Zampetakis*
TIBCO Orchestra Networks
zabetak@gmail.com

## ABSTRACT

Big data applications routinely involve diverse datasets: relations flat or nested, complex-structure graphs, documents, poorly structured logs, or even text data. To handle the data, application designers usually rely on several data stores used side-by-side, each capable of handling one or a few data models, and each very efficient for some, but not all, kinds of processing on the data. A current limitation 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*) partial query results (in the style of materialized views) which may be available in the stores.

We present ESTOCADA, a novel approach connecting applications to the potentially heterogeneous systems where their input data resides. ESTOCADA 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. ESTOCADA 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. Our experiments illustrate the significant performance gains achieved by ESTOCADA.

## CCS CONCEPTS

• **Database management systems → Polystore systems**.

## KEYWORDS

cross-model query rewriting, integrity constraints, query optimization

## 1 INTRODUCTION

Big Data applications increasingly involve *diverse* data sources, such as: flat or nested relations, structured or unstructured documents, data graphs, relational databases, 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., relational data gets loaded into a relational or "Big Table"-style system, JSON documents in a JSON store, etc. Heterogeneous stores also "accumulate" in an application along the time, e.g., at some point one decision is made to host dataset $D_1$ in Spark and $D_2$ in MongoDB, while later on another application needs to use $D_1$ and $D_2$ together. Systems capable of exploiting diverse Big Data in this fashion are usually termed *polystores* [7, 20, 21, 39]. Query evaluation in a polystore recalls to some extent mediator systems [31]; in both cases, sub-queries are delegated to the underlying stores when possible, while the remaining operations are applied in the upper integration layer. Current polystores process a query assuming that each of its input datasets is available in exactly one store (often chosen for its support of a given 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 partially 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 propose 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;* further, 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.* Importantly, our approach *does not require any change to the application code.* The example below illustrates these performance-enhancing opportunities.

### 1.1 Motivating example

Consider the Medical Information Mart for Intensive Care III (MIMIC-III) [36] dataset, comprising health data for more than 40.000 Intensive Care Unit (ICU) patients from 2001 to 2012. The total size of **46.6 GB**, and it consists of JSON documents describing: (*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 (e.g., MD stands for "medical doctor"), (*iii*) lab measurements (e.g., ABG stands for "arterial blood gas") and (*iv*) diagnosis related groups (DRG) codes descriptions.

**Our motivation query** $Q_1$ is: "for the patients transferred into the ICU due to "coronary artery" issues, with abnormal blood test

*Work done while the author was a PhD student working at Inria, France.

results, find the date/time of admission, their previous location (e.g., emergency room, clinic referral), and the drugs of type "additive" prescribed to them". Evaluating this query through the AsterixDB [3] (Version 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), denoted Postgres in the sequel, $Q_1$ took 12.6 minutes.

Now, consider we had at our disposal three *materialized views* which pre-compute partial results for $Q_1$. SOLR[1] is a well-known highly efficient full-text server, also capable of handling JSON documents. Consider a SOLR server stores a view $V_1$ storing the IDs of patients and their hospital admission data, as well as the caregivers' reports, include notes on the patients' stay (e.g., detailed diagnosis, etc). Full-text search on $V_1$ for "coronary artery" allows to efficiently retrieve the IDs of the respective patients. Further, consider that a Postgres server stores a view $V_2$ with the patients meta-data information and their hospital admission information such as admission time and patients' location prior to admission. Finally, assume available a view $V_3$ which stores all drugs that are prescribed for each patient that has "abnormal blood" test results, as a JSON document stored in Postgres.

Now, we are able to evaluate $Q_1$ by a full-text search on $V_1$ followed by a BindJoin [51] 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 simple Java-based execution engine (implementing select, project, join, etc.) to access the views and join them, this takes about 5.70 mins, or a **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 drastically 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, Postgres' efficient join algorithms, etc.) is the second reason (together with materialized view usage) for our 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 [31], 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*, e.g., MIMIC is in JSON, Apache log data is often represented in CSV, etc. 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.

[1] http://lucene.apache.org/solr

| Data model | Query language/API | Systems |
|---|---|---|
| Relational | SQL | Major vendors |
| JSON | SparkSQL | Spark [11] |
| JSON | AQL/SQL++ | AsterixDB [3] |
| JSON | SQLw/JSON | Postgres |
| Key-value | Redis API | Redis |
| Full-text and JSON | Solr API | Solr |
| XML | XQuery, XPath | Saxon [4] |

**Table 1: Data Stores Supported by ESTOCADA**

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 [42], or XML [13, 46, 49]. Different from these, some LAV systems [18, 44] 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*. In this paper, we introduce a novel approach for cross-model view-based rewriting and we report on its implementation and experimental evaluation in a polystore context. Our approach is currently capable of handling the systems listed in Table 1, together with their data models and query languages.

## 1.2 Outline

The remainder of this paper is organized as follows. Section 2 formalizes the rewriting problem we address in this paper. Section 3 outlines our approach; at its core is a view-based query rewriting algorithm based on an internal model, invisible to users and applications, but which crucially supports rewriting under integrity constraints. Section 4 describes the language we use for (potentially cross-model) views and queries. Section 5 shows how we reduce the multi-data model rewriting problem to one within this internal pivot model, then transform rewritings obtained there into data integration plans. Section 6 describes a set of crucial optimizations and extensions we contributed to the rewriting algorithm at the core of our approach, in order to make it scale to our polystore setting; we formalize the guarantees on this algorithm in Section 7. We present our experimental evaluation in Section 8, discuss related work in Section 9 then conclude in Section 10.

# 2 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; for instance, AsterixDB, MongoDB and Postgres have different query languages for JSON. As we shall see, we consider

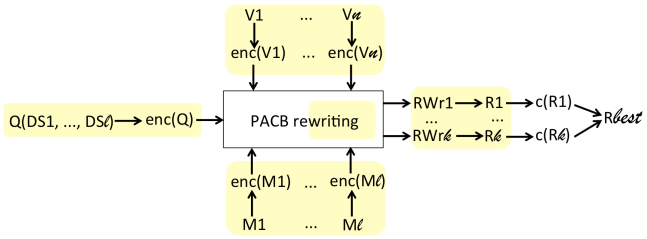


**Figure 1: Outline of our approach**

*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 $S$ can store instances of the model $M_S$ and evaluate queries expressed in $L_S$.

We consider a set of data sets $\mathcal{D} = \{D_1, D_2, \ldots\}$, such that each data set $D \in \mathcal{D}$ is an instance of a data model $M_D$. A data set $D$ 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 $S_D^j$ [2].

We consider an *integration language* $\mathcal{IL}$, 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.

Such integration language has been proposed in several works [20, 39]; we use it to express computations over the views. An $\mathcal{IL}$ expression bears obvious similarities with an execution plan in a wrapper-mediator system; its evaluation is split between computations pushed to the stores, and subsequent operations applied by the mediator on top of their results. The main distinction is that $\mathcal{IL}$ remains declarative, abstracting the details of each in-store computation, which is simply specified by a query in the store's language.

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). The cost may reflect e.g., disk I/O, memory needs, CPU time, transfer time between distributed sites etc. We outline the concrete model we use in Appendix K.

We term *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 $c(r)$.*

Most modern query languages include such primitives as arithmetic operations, aggregation, and calls to arbitrary UDFs; these suffice to preclude decidability of checking whether two queries are equivalent. We therefore aim at finding rewritings *under black-box (uninterpreted) function semantics (UFS)*: we seek rewritings that make the same function calls, with the same arguments as the original query.

## 3 OVERVIEW AND ARCHITECTURE

We outline here our approach for solving the above problem.

**Our integration language: $QBT^{XM}$.** We devised a concrete integration language, called $QBT^{XM}$, which supports queries over several data stores, each with its own data model and query language. $QBT^{XM}$ follows a *block*-based design, with blocks organized into a tree in the spirit of the classical Query Block Trees (QBT) introduced in System R [52], slightly adapted to accommodate subsequent SQL developments, in particular, the ability to nest subqueries within the select clause [34]. The main difference in our setting is that each block may be expressed in a different query language and carry over data of a different data model (e.g., SQL for relational data, key-based search API for key-value data, different JSON query languages etc.) We call the resulting language $QBT^{XM}$, for *cross-model QBT* (detailed in Section 4.2); excerpts of its grammar are delegated to Appendix I.

**$QBT^{XM}$ views.** Each materialized view $V$ is defined by an $QBT^{XM}$ 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.

**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 also Figure 1; yellow background identifies the areas where we bring contributions in this work.). First, we *encode relationally* the structure of original data sets, the view specifications and the application query.

Note that the relations used for 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 using 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). For instance, we describe the organization of a JSON document data model using a small set of relations such as *Node(nID, name)*, *Child(pID, cID)*, *Descendant(aID, dID)*, etc. together with the constraints specifying that every node has just one parent and one tag, that every child is also a descendant, etc. Such modeling had first been introduced in local-as-view XML integration works [18, 44]. The constraints are tuple-generating dependencies (*tgds*) or equality-generating dependencies (*egds*) [6], extended with such well-researched constraints as denial constraints [28] and disjunctive constraints [27]. We detail the pivot model in Section 5.1.

**Reduction 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 the view's input data and its output. Constraints have been used to encode single-model views [19] and correspondences between source and target schemas in data exchange [23]. The novelty here is the

[2] 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.

rich collection of supported models, and the cross-model character of the views.

An incoming query $Q$ over the original datasets $DS_1, \ldots, DS_l$, whose data models respectively are $M_1, \ldots, M_l$, is encoded as a relational query $enc(Q)$ over the dataset's relational encoding. $enc(Q)$ is centered around conjunctive queries, with extensions such as aggregates, UDFs, nested sub-queries, disjunction and negation.

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 $QBT^{XM}$ 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 $V_1, \ldots, V_n, M_1, \ldots, M_l$. The reduction is detailed in Sections 5.2 and 5.3.

**Relational rewriting using constraints.** To solve the single-model rewriting problem, we need to reformulate relational queries under constraints. The algorithm of choice is known as Chase & Backchase (C&B, in short), introduced in [17] and improved in [33] to yield the *Provenance-Aware C&B algorithm* (PACB, in short). PACB was designed to work with relatively few views and constraints. In contrast, in the polystore setting, each of the many datasources is described by a view, and each view is encoded by many constraints. For instance, the JSON views in our experiments (Section 8) are encoded via ~ 45 tgds involving 10-way joins in the premise. To cope with this complexity, we modified PACB to incorporate novel scale-up techniques (discussed in Section 6.1). PACB was designed for conjunctive queries under set semantics. Towards supporting a richer class of queries, we extended it to bag semantics (Section 6.2) and QBTs (Section 6.3).

**Decoding the relational rewritings.** On each relational reformulation $RW_r^i$ issued by our modified PACB rewriting, a *decoding step* $dec(RW_r^i)$ is performed to:

(*i*) Group the reformulation atoms by the view they pertain to; for instance, we infer that the three atoms *Document(dID, "file.json"), Root(dID, rID), Child(rID, cID), Node(cID, book)* refer to a single JSON document by following the connections among nodes and knowledge of the JSON data model.

(*ii*) Reformulate each such atom group into a query which can be completely evaluated over a single view.

(*iii*) If several views reside in the same store, identify the largest subquery that can be delegated to that store, along the lines of query evaluation in wrapper-mediator systems [29].

**Evaluation of non-delegated operations.** A decoded rewriting may be unable to push (delegate) some query operations to the store hosting a view, if the store does not support them; for instance, most key-value and document stores do not support joins. Similarly, if a query on structured data requests the construction of new nested results (e.g., JSON trees) and if the data is in a store that does not support such result construction natively, it will have to be executed outside of that store. To evaluate such "last-step" operations, we rely on a *lightweight execution engine* [1], which uses a nested relational model and supports bind joins [51] (with sideways information passing) to evaluate nested subqueries. Our engine can be easily replaced with another similar execution engine; we exemplify this with BigDAWG [20] in Section 8.1.2.

**Choice of the most 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. The problem of choosing the best rewriting and best evaluation plan in this setting recalls query optimization in distributed mediator systems [47]. 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. While devising cost models for polystore settings is beyond the scope of this paper, we architected ESTOCADA to take the cost model as configuration parameter. For our experiments we designed a simple cost model (detailed in Appendix K) that suffices to show the benefit of our approach.

# 4 THE $QBT^{XM}$ LANGUAGE

We present here the language we use for views and queries. First, we recall the classical Query Block Trees (QBTs), then we extend them to cross-model views and queries.

## 4.1 Query Block Trees (QBT)

Since many of the languages ESTOCADA supports allow for nested queries and functions (user-defined or built-in such as aggregates), our pivot language *Query Block Trees* (QBTs) also supports these features. These are essentially the classical System R QBTs [52], slightly adapted to accommodate subsequent SQL extensions, such as nesting in the select clause (introduced in SQL-1999 [34]). We first illustrate QBTs on a SQL example.

**Example** 1. *Consider the following SQL query which computes, for each student who is not enrolled in any course for the Spring'16 quarter, the number of Spring'16 courses she is waitlisted for (a count of* 0 *is expected if the student is not waitlisted). While this query could be written using joins, outer joins and group by operations, we show a natural alternative featuring nesting (which illustrates how we systematically normalize away such operations):*

```
SELECT s.ssn, COUNT (SELECT w.cno
                     FROM   Waitlist w
                     WHERE  w.ssn = s.ssn
                            AND w.qtr = 'Spring 2016'
                            ) AS cnt
FROM    Student s
WHERE   NOT EXISTS (SELECT  c.no
                    FROM    Course c, Enrollment e
                    WHERE   c.no = e.cno
                            AND s.ssn = e.ssn
                            AND e.qtr = 'Spring 2016')
```

*This query consists of three blocks. The outermost SELECT-FROM-WHERE expression corresponds to the root block, call it $B_0$. The two nested SELECT-FROM-WHERE expressions are modeled as children blocks of the root block, call them $B_{00}$ and $B_{01}$ for the block nested in the SELECT clause, respectively the block nested in the WHERE clause.* □

The variables occurring in a block $B$ can be either defined by $B$ (in which case we call them *bound* in $B$), or by one of its ancestors (in which case they are called *free* in $B$). We assume w.l.o.g. that the bound variables of $B$ have fresh names, i.e., they share no names with variables bound in $B$'s ancestors.

**Example** 2. *In Example 1, variable s is bound in $B_0$, but it occurs free in both $B_{00}$ and $B_{01}$.* □

QBTs model select-from-where expressions as blocks, organized into a tree whose parent-child relationship reflects block nesting. Nested blocks always appear as arguments to functions, be they built-in (e.g., COUNT for $B_{00}$ and NOT EXISTS for $B_{01}$ in Example 1) or user-defined. While not shown in Example 1, the case of blocks nested within the FROM clause corresponds to the identity function.

As we will show, we use QBTs to translate relationally application queries written in other models, but also (potentially cross-model) views and rewritings.

## 4.2 Integration Language: $QBT^{XM}$

To specify cross-model queries and views, we adopted a *block*-based design, similar to QBTs, but in which each block is expressed in its own language, signaled by an annotation on the block. We call the resulting language $QBT^{XM}$, which stands for *cross-model QBT*. $QBT^{XM}$ queries comprise a FOR and a RETURN clause. The FOR clause introduces several variables and specifies their legal bindings. The RETURN clause specifies what data should be constructed for each binding of the variables. Variables can range over different data models, which is expressed by possibly several successive blocks, each pertaining to its own model. In SQL style, this defines a Cartesian product of the variable bindings computed by each block from the FOR clause; this Cartesian product can be filtered through WHERE clause. We impose the restriction that *variables shared by blocks spanning different data models must be of text or numeric type*, so as to avoid dealing with conversions of complex values across models. While there is no conceptual obstacle to handle such conversions automatically, the topic is beyond the scope of this paper.

We describe $QBT^{XM}$ informally, by examples; the grammar of $QBT^{XM}$ is delegated to Appendix I.

**Example** 3. *We illustrate the $QBT^{XM}$ definition of views $V_1$ and $V_2$ from Section 1, using AsterixDB's SQL++ syntax (easier to read than the JSON syntax of Postgres):*

```
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}
View V2:
FOR AJ:{SELECT  M.patientID AS patientID,
                A.admissionID AS admissionID,
                A.admissionLoc AS admissionLoc
                A.admissionTime AS admissionTime
         FROM  MIMIC M, M.admissions A}
RETURN PR:{patientID,admissionID,
           admissionLoc,admissionTime}
```

□

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, PR and SJ stand for the SQL++ language of AsterixDB and the languages of Postgres and Solr, respectively. Also, below, PJ and RK stand respectively for Postgres' JSON query language, and for a simple declarative key-value query language we designed for Redis.

# 5 REDUCTION FROM CROSS-MODEL TO SINGLE-MODEL SETTING

A key requirement in our setting is the ability to reason about queries and views involving multiple data models. This is challenging for several reasons. First, not every data model/query language setting supported in ESTOCADA comes with a known *view-based query rewriting (VBQR, in short) algorithm*: consider key-value pairs as data model and declarative languages over them, or JSON data and its query languages in circulation, etc. Second, even if we had these algorithms for each model/language pair, neither would readily extend to a solution of the *cross-model* VBQR setting in which views may be defined over data from various models, materializing their result in yet again distinct models, and in which query rewritings access data from potentially different models than the original query. Third, for the feasibility of development and maintenance as the set of supported model/language pairs evolves, any cross-model VBQR solution needs to be modularly extensible to additional models/languages, in the sense that no modification to the existing code is required and it suffices to just add code dealing with the new model/language pair. Moreover, the developer of this additional code should not need to understand any of the other model/language pairs already supported in ESTOCADA.

To address these challenges, we ***reduce the cross-model VBQR problem to a single-model VBQR problem***. That is, we propose a unique *pivot data model* (Section 5.1), on which QBT (Section 4.1) serves as a *pivot query language*. Together, they allow us to capture data models, queries and views so that cross-model rewritings can be found by searching for rewritings in the single-model pivot setting instead. Section 5.1 presents the pivot model; then, Section 5.2 presents query encoding in the pivot language, Section 5.3 shows how to encode views as constraints, finally Section 5.4 describes the decoding of rewriting into integration plans.

## 5.1 The Pivot Model

Our pivot model is relational, and it makes prominent use of expressive integrity constraints. This is sufficiently expressive to capture the key aspects of the data models in circulation today, including: freely nested collections and objects, object identity (e.g., for XML element nodes and JSON objects), limited binding patterns (as required by key-value stores and full-text indexes such as Lucene/Solr), relationships (in the E/R and ODL sense), functional dependencies and much more. The integrity constraints we use are tuple-generating dependencies (tgds) or equality-generating dependencies (egds) [6], extended to denial constraints [28] and disjunctive constraints [27].

**JSON.** We represent JSON documents as relational instances conforming to the schema VREJ (Virtual Relational Encoding of JSON) in Table 2 below. We emphasize that these relational instances are *virtual*, i.e., the JSON data is not stored as such. Regardless of the JSON data's physical storage, we only use VREJ to encode JSON

queries relationally, in order to reason about them.

| $Collection_J(name, id)$ |
|---|
| $Child_J(parentId, childId, key, type)$ |
| $Eq_J(x, y)$ |
| $Value_J(x, y)$ |

**Table 2: Snippet of VREJ**

VREJ constraints express the fact that JSON databases are organized into named collections of JSON values, which in turn can be objects (consisting of a set of key-value pairs), arrays of values, and scalar values (numeric or text).

We model objects, arrays and values in an object-oriented fashion, i.e. they have identities. This is because some JSON stores, e.g., MongoDB and AsterixDB, support query languages that refer to identity and/or distinguish between equality by value versus equality by identity. Our modeling supports both the identity-aware and the identity-agnostic view of JSON, via appropriate translation of queries and views into pivot language expressions over the VREJ schema. Relation $Collection_J$ attaches to each persistent name the id of an array. $Value_J$ attaches a scalar value to a given id. $Child_J$ states that the value identified by *childId* is immediately nested within the value identified by *parentId*. The *type* attribute records the type of the parent (array "a" or object "o") and determines the kind of nesting and the interpretation of the *key* attribute: if the parent is an array, *key* holds the position at which the child occurs; if the parent is an object, then *key* holds the name of the key whose associated value is the child.

Our modeling of parent-child (immediate nesting) relationship between JSON values provides the type of value *only in conjunction with a navigation step where this value is a parent*, and the step leads to a child value of undetermined type. This modeling reflects the information one can glean statically (at query rewriting time, as opposed to query runtime) from JSON query languages in circulation today. Recall that JSON data is semi-structured, i.e., it may lack a schema. Queries can, therefore, express navigation leading to values whose type cannot be determined statically if these values are not further navigated into. The type of a value can be inferred only from the type of navigation step into it.

**Example** 4. *If a query navigation starts from a named JSON collection "coll" and proceeds to the fifth element e therein, we can infer that "coll" is of array type, but we do not know the type of e. Only if the query specifies an ensuing lookup of the value v associated to key "k" in e can we infer e's type (object). However, absent further navigation steps, we cannot tell the type of the child value v.* □

Finally, relation $Eq_J$ is intended to model value-based equality for JSON (id-based equality is captured directly as equality of the id attributes).

We capture the intended meaning of the VREJ relations via constraints that are inherent in the data model (i.e. they would hold if we actually stored JSON data as a VREJ instance). We illustrate a few of these below; following common practice, free variables are to be read as universally quantified.

The fact that a persistent name uniquely refers to a value is expressed by the egd (1) below. Egd (2) states that an object cannot have two distinct key-value pairs sharing the same key, or that an array cannot have two distinct elements at the same position. Tgds (3) and (4) state that value-based equality is symmetric, respectively transitive, and tgd (5) states that value-equal parents must have value-equal children for each parent-child navigation step. Egd (6) states that no two distinct scalar values may correspond to a given id.

$$Collection_J(n,x) \wedge Collection_J(n,y) \rightarrow x = y \quad (1)$$

$$Child_J(p,c_1,k,t) \wedge Child_J(p,c_2,k,t) \rightarrow c_1 = c_2 \quad (2)$$

$$Eq_J(x,y) \rightarrow Eq_J(y,x) \quad (3)$$

$$Eq_J(x,y) \wedge Eq_J(y,z) \rightarrow Eq_J(x,z) \quad (4)$$

$$Eq_J(p,p') \wedge Child_J(p,c,k,t) \rightarrow \exists c'\ Eq_J(c,c') \wedge Child_J(p',c',k,t) \quad (5)$$

$$Value_J(i,v_1) \wedge Value_J(i,v_2) \rightarrow v_1 = v_2 \quad (6)$$

**Key-Value Store Model.** Our interpretation of the key-value pairs model is compatible with many current systems, in particular Redis, supported by ESTOCADA. Calling a *map* a set of key-value pairs, the store accommodates a set of persistently named maps (called *outer* maps). Within each outer map, the value associated to a key may be either itself a map (called an *inner* map), or a scalar value. In case of an *inner* map, the value associated to a key is a scalar value.

Given the analogy between key-value and JSON maps, we model the former similarly to the latter, as instances of the relational schema VREK, consisting of the relations: $Map_{KV}(name, mapId)$, $Child_{KV}(parentId, childId, key, type)$ and $Eq_{KV}(x,y)$. Here, $Map_{KV}$ models the association between a persistent name and (the id of) an outer map. $Child_{KV}$ reflects the immediate nesting relationship between a map (the parent) and a value associated to one of its keys (the child). The *type* attribute tells us whether the parent map is an outer or an inner map (these are treated asymmetrically in some systems). The $Eq_{KV}$ relation models value-based equality, analogously to $EQ_J$ for JSON.

The intended semantics of these relations is enforced by similar constraints to the JSON model, e.g., in a map, there is only one value for a key ($Child_{KV}(p,c_1,k,t), Child_{KV}(p,c_2,k,t) \rightarrow c_1 = c_2$), persistent map names are unique ($Map_{KV}(n,x), Map_{KV}(n,y) \rightarrow x = y$) etc.

**XML.** Query reformulation for XML has already been reduced in prior work [18, 19] to a purely relational problem. An XML document was represented as a *virtual* instance of the relational schema *VREX* consisting of the relations: {*Root*, *Elem*, *Child*, *Desc*, *Tag*, *Attrib*, *Id*, *Text*}.

The intended meaning of the relations in VREX reflects the fact that XML data is a tagged tree. The unary predicate *Root* denotes the root element of the XML document, and the unary relation *Elem* is the set of all elements. *Child* and *Desc* are subsets of $Elem \times Elem$ and they say that their second component is a child, respectively a descendant of the first component. $Tag \subseteq Elem \times string$ associates the tag in the second component to the element in the first. $Attrib \subseteq Elem \times string \times string$ gives the element, attribute name and attribute value in its first, second, respectively third component. $Id \subseteq string \times Elem$ associates the element in the second component to a string attribute in the first that uniquely identifies it (if DTD-specified ID-type attributes exist, their values can be used for this). $Text \subseteq Elem \times string$ associates to the element in its first component the string in its second component. [19] shows that some of the

intended meaning of schema VREX is captured by a set *TIX* (*T*rue *I*n *X*ML) of constraints expressible as tgds and egds, in some cases extended with disjunction. These constraints state, for instance, that the root of the XML tree is unique (expressible as an egd), and that every element has precisely one tag (an egd and a tgd). They also state that *Desc* is reflexive and transitive (tgds). The tree shape of the *Child* relation is expressed by constraints stating that every element has at most one parent (an egd), at least one tag (a tgd), at most one tag (egd), etc.

**Relational.** It is well known that the relational data model endowed with key, uniqueness and foreign key constraints is captured by our pivot model: key/uniqueness constraints are expressible by egds, and foreign key constraints by tgds.

**Binding Patterns.** We have seen above a natural way to model sets of key-value pairs using a relation. To simplify the discussion, we abstract from the *parentId* and *type* attributes of relation $Child_{KV}$ above, focusing on the binary relationship between keys and values: $KV(key, value)$. Note that typical key-values store APIs require that values can be looked up only given their key, but not conversely. If we do not capture this limitation, the rewriting algorithm may produce rewritings that correspond to no executable plan. For instance, consider a rewriting of the form: $R(v) : -KV(k, v)$. $R$ corresponds to no legal plan given the access limitation of KV, because $R$ requires the direct extraction of all values stored in the store, without looking them up by key.

This setting involving relations with limited lookup access is well-known in the literature, and is modeled via the concept of relations adorned with *binding patterns* [51].

In a relational setting, query rewriting when access to the views is limited by binding patterns has been studied for conjunctive queries and views [25, 45], yet complete rewriting under both integrity constraints and binding patterns was not considered until [15]. [15] shows how to encode binding patterns using tgd constraints, which fit into our pivot model. The idea is to introduce a unary relation, say $A(x)$ to state that $x$ is accessible, and for each binding pattern a tgd stating that when all required input attributes of a relation are accessible, then the output attributes are accessible as well. This allows the chase with such tgds to determine which attributes of a rewriting are accessible.

**Nulls, Equality and Aggregation.** Today's crowded landscape of stores exhibits wide variability in the treatment of nulls [48], thus equality and aggregation semantics may differ across stores. To account for this, by default we use distinct null constants, distinct aggregate functions and distinct equality relationships for the relational encoding of distinct stores. For instance, the sum aggregate of language $\mathcal{L}_1$ is not automatically equated to the sum of $\mathcal{L}_2$, since the two may treat nulls differently. When the null treatment coincides totally (e.g., for distinct but SQL-compliant RDBMS stores), we use a single common encoding. When it coincides only partially, we encode as much as possible with constraints. For instance the tgd $eq_1(null_1, x) \rightarrow eq_2(null_2, x)$ states that if a value $x$ is equal to null in System 1's interpretation of null and equality, this holds also in System 2's interpretation.

```
FOR AJ:{SELECT M.patientID AS patientID,
               A.admissionLoc AS admissionID,
               A.admissionTime AS admissionTime,
               P.drug AS drug
        FROM   LABITEM L, MIMIC M, M.admissions A,
               A.labevents LE, A.prescriptions P
        WHERE  L.itemID=LE.labItemID AND
               L.category='blood' AND
               L.flag ='abnormal' AND
               P.drugtype='additive' AND
               contains(report,'coronary artery')}
RETURN AJ:{"patientID":patientID, "drug":drug,
           "admissionLoc":admissionLoc,
           "admissionTime":admissionTime}
```

**Figure 2: Motivating scenario $QBT^{XM}$ Query $Q_1$**

```
MIMIC(M),
Child_J(M,patientID,"patientID","o"),
Child_J(M,A,"admissions","o"),
Child_J(A,admissionID,"admissionID","o"),
Child_J(A,admissionLoc,"admissionLoc","o"),
Child_J(A,admissionTime,"admissionTime","o"),
Child_J(A,report,"reports","o")->
V_2(patientID,admissionID,admissionLoc,admissionTime)
```

**Figure 3: Relational encoding of $QBT^{XM}$ View $V_2$**

## 5.2 Encoding $QBT^{XM}$ Queries into the Pivot Language

The purpose of the pivot language is to reduce the VBQR problem to a single-model setting. The pivot model enables us to represent relationally queries expressed in the various languages supported in our system (recall Table 1), and which can be combined into a single $QBT^{XM}$ query. As shown in Figure 1, an incoming query $Q$ is *encoded* as a relational conjunctive $enc(Q)$ over the relational encoding of the datasets it refers to (with extensions such as aggregates, user-defined functions, nested queries, disjunction and negation). Figure 2 shows the $QBT^{XM}$ query $Q_1$ of the motivating scenario, and its relational encoding $enc(Q_1)$ appears in Appendix A.

## 5.3 Encoding $QBT^{XM}$ Views as Constraints

We translate each view definition $V$ into additional relational integrity constraints $enc(V)$ showing how the view inputs are related to its output. Figure 3 illustrates the relational encoding of $QBT^{XM}$ view $V_2$ from Section 4 (for space reasons, similar constraints resulting from $V_1$ and $V_2$ appear in Appendix B. Due to space limitation, we omit the constraints for $V_3$ from this version of the paper).

## 5.4 From Rewritings to Integration Plans

We translate each query rewriting into a *logical plan* specifying (*i*) the (native) query to be evaluated within each store, and (*ii*) the remaining logical operations, to be executed within the integration engine. The translation first decodes $RW_r$ into $QBT^{XM}$ syntax. Based on the heuristic that the store where a view resides is more efficient than the mediator, and thus it should be preferred for all the operations it can apply, decoding tries to delegated to the stores as much as possible. To that end, it first partitions each rewriting $RW_r$ into view-level subqueries, which are sets of atoms referring to the virtual relations of a *same view*. If a group of view-level subqueries pertains to the same store and if the store supports joins, we translate the group to a single query to be pushed to the store.

**Example** 5 (Delegation-Aware Decoding). *Consider the rewriting $RWQ_1$ of $QBT^{XM}$ query $Q_1$ shown in Figure 4. First, we delimit the view-wide subqueries (delimited by empty lines in the figure), calling them $RWQ_1^1, RWQ_1^2$ and $RWQ_1^3$ in order from the top. The*

```
RWQ1<patientID,drug,admissionLoc,admissionTime>:-
V1(d1),
Child_JV1(d1,patientID,"patientID","o"),
Child_JV1(d1,admissionID,"admissionID","o"),
Child_JV1(d1,report,"report","o"),
  Value_J(report,"like-coronary artery"),

V2(patientID,admissionID,admissionLoc,admissionTime),

V3(d2),
Child_JV3(d2,patientID,"patientID","o"),
Child_JV3(d2,A,"admission","o"),
Child_JV3(A,admissionID,"admissionID","o"),
Child_JV3(A,drugs,"drugs","o"),
Child_JV3(drugs,drug,"drug","o"),
Child_JV3(drugs,drugtype,"drugtype","o"),
    Value_J(drugtype,"additive")
```

**Figure 4: Relational rewriting $RWQ_1$ of $QBT^{XM}$ query $Q_1$**

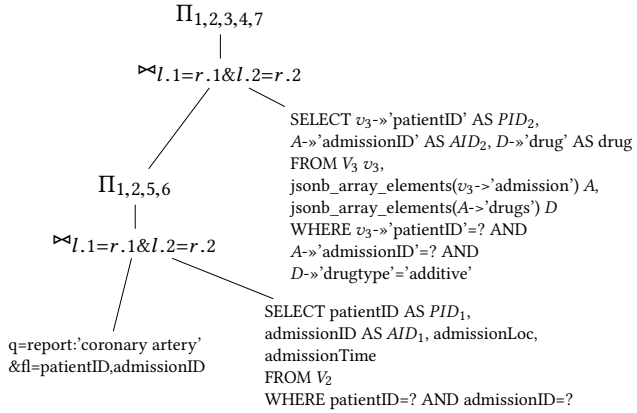


**Figure 5: Integration plan of the motivating scenario**

*subquery heads contain variables from the head of $RWQ_1$, as well as join variables shared with other subqueries. For example, the head of $RWQ_1^2$ contains the variables admissionLoc and admissionTime (in the head of $RWQ_1$), and also patientID, admissionID, needed to join with $RWQ_1^1$, and $RWQ_1^3$. These relational subqueries are then decoded to the native syntax of their respective stores, each constituting a block of the resulting $QBT^{XM}$ rewriting $dec(RWQ_1)$ (shown in Appendix C).* □

The ESTOCADA plan generator next translates the decoded $QBT^{XM}$ rewriting to a logical plan that pushes leaf blocks to their native store, applying last-step operators on the results.

The integration plan for $RWQ_1$ is shown in Figure 5.

## 6 PACB OPTIMIZATION AND EXTENSION

The state-of-the-art algorithm for view-based rewriting of relational queries under relational constraints is known as PACB [33]. We detail below just enough of its inner working to explain our optimization. Then, Section 6.1 introduces our optimization in the original PACB setting (conjunctive relational queries and set semantics). Section 6.2 extends this to bag semantics, Section 6.3 extends it to $QBT^{XM}$.

A key ingredient of the PACB algorithm is to *capture views as constraints*, in particular tgds, thus reducing the view-based rewriting problem to constraints-only rewriting. For a given view $V$, the constraint $V_{IO}$ states that for every match of the view body against the input data there is a corresponding tuple in the view output; the constraint $V_{OI}$ states the converse inclusion, i.e., each view tuple is due to a view body match. Then, given a set $\mathcal{V}$ of view definitions, PACB defines a set of view constraints $C_\mathcal{V} = \{V_{IO}, V_{OI} \mid V \in \mathcal{V}\}$.

The constraints-only rewriting problem thus becomes: given the source schema $\sigma$ with a set of integrity constraints $\mathcal{I}$, a set $\mathcal{V}$ of view definitions over $\sigma$ and the target schema $\tau$ which includes $\mathcal{V}$, given a conjunctive query $Q$ expressed over $\sigma$, find reformulations $\rho$ expressed over $\tau$ that are equivalent to $Q$ under the constraints $\mathcal{I} \cup C_\mathcal{V}$.

For instance, if $\sigma = \{R,S\}$, $\mathcal{I} = \emptyset$, $\tau = \{V\}$ and view $V$ materializes the join of tables $R$ and $S$, $V(x,y) :- R(x,z), S(z,y)$, the constraints capturing $V$ are:

$$V_{IO}: \quad R(x,z) \wedge S(z,y) \rightarrow V(x,y)$$
$$V_{OI}: \quad V(x,y) \rightarrow \exists z\, R(x,z) \wedge S(z,y).$$

For input $\sigma$-query $Q(x,y) :- R(x,z), S(z,y)$, PACB finds the $\tau$-reformulation $\rho(x,y) :- V(x,y)$. Algorithmically, this is achieved by:

($i$) chasing $Q$ with the set of constraints $\mathcal{I} \cup C_\mathcal{V}^{IO}$ where $C_\mathcal{V}^{IO} = \{V_{IO} \mid V \in \mathcal{V}\}$;

($ii$) restricting the chase result to only the $\tau$-atoms (the result is called the *universal plan*) $U$;

($iii$) chasing $U$ with the constraints in $\mathcal{I} \cup C_\mathcal{V}^{OI}$, where $C_\mathcal{V}^{OI} = \{V_{OI} \mid V \in \mathcal{V}\}$; the result is denoted $B$ and called the *backchase*; finally:

($iv$) matching $Q$ against $B$ and outputting as rewritings those subsets of $U$ that are responsible for the introduction (during the backchase) of the atoms in the image of $Q$. [3]

In our example, $\mathcal{I}$ is empty, $C_\mathcal{V}^{IO} = \{V_{IO}\}$, and the result of the chase in phase ($i$) is $Q_1(x,y) :- R(x,z), S(z,y), V(x,y)$. The universal plan obtained in phase ($ii$) by restricting $Q_1$ to $\tau$ is $U(x,y) :- V(x,y)$. The result of backchasing $U$ with $C_\mathcal{V}^{OI}$ in phase ($iii$) is $B(x,y) :- V(x,y), R(x,z), S(z,y)$ and in phase ($iv$) we find a match from $Q$'s body into the $R$ and $S$ atoms of $B$, introduced during the backchase due to $U$'s atom $V(x,y)$. This allows us to conclude that $\rho(x,y) :- V(x,y)$ is an equivalent rewriting of $Q$.

### 6.1 PACB$^{\mathbf{opt}}$: Optimized PACB

The idea for our optimization was sparked by the following observation. The backchase phase (step ($iii$)) involves the $V_{OI}$ constraints for all $V \in \mathcal{V}$. The backchase attempts to match the left-hand side of each $V_{OI}$ for each $V$ repeatedly, leading to wasted computation for those views that have no match. In a polystore setting, the large number of data sources and stores lead to a high number of views, most of which are often irrelevant for a given query $Q$.

This observation leads to the following modified algorithm. Define the set of views mentioned in the universal plan $U$ as *relevant*

[3] Recall that a chase step $s$ with constraint $c$ matches $c$'s premise against existing atoms $e$ and adds new atoms $n$ corresponding to $c$'s conclusion. To support fast detection of responsible atoms in Phase ($iv$), $s$ records that the $e$ atoms are responsible for the introduction of the $n$ atoms [33]. Our optimization does not affect Phase ($iv$).

*to $Q$ under $\mathcal{I}$*, denoted $relev_{\mathcal{I}}(Q)$. Define the optimized PACB algorithm PACB$^{\text{opt}}$ identical with PACB except phase (*iii*) where PACB$^{\text{opt}}$ replaces $C_{\mathcal{V}}^{OI}$ with $C_{relev_{\mathcal{I}}(Q)}^{OI}$. That is, PACB$^{\text{opt}}$ performs the backchase only with the $OI$-constraints of the views determined in phase (*i*) to be relevant to $Q$, ignoring all others. This modification is likely to save significant computation when the polystore includes many views. In our example assume that, besides $V$, $\mathcal{V}$ contained 1000 other views $\{V^i\}_{1\leq i\leq 1000}$, each irrelevant to $Q$. Then the universal plan obtained by PACB would be the same as $U$ above, and yet $\{V_{OI}^i\}_{1\leq i\leq 1000}$ would still be considered by the backchase phase, which would attempt at every step to apply each of these 1000 constraints. In contrast, PACB$^{\text{opt}}$ would save this work.

In general, ignoring even a single constraint $c$ during backchase may lead to missed rewritings [50]. This may happen *even when $c$ mentions a part of the schema $\sigma$ disjoint from the $\sigma$ part mentioned in $U$*, if the backchase with $\mathcal{I}$ exposes semantic connections between these disjoint schema parts. In our setting, we prove that this is not the case:

Theorem 6.1. *Algorithm PACB$^{opt}$ finds the exact same rewritings as the original PACB.*

This is because we only ignore some *view-capturing constraints*, which can be shown (by analysing the backchase evolution) *never* to apply, regardless of the constraints in $\mathcal{I}$. Combined with the main result from [33], Theorem 6.1 yields the completeness of PACB$^{\text{opt}}$ in the following sense:

Corollary 6.2. *Whenever the chase of input conjunctive query $Q$ with the constraints in $\mathcal{I}$ terminates*[4]*, we have:*

*(a) algorithm PACB$^{opt}$ without a cost model enumerates* all join-minimal[5] *$\mathcal{V}$-based rewritings of $Q$ under $\mathcal{I}$, and*

*(b) algorithm PACB$^{opt}$ equipped with a cost model $c$ finds the $c$-optimal join-minimal rewritings of $Q$.*

In Section 8.2, we evaluate the performance gains of PACB$^{\text{opt}}$ over the original PACB, measuring up to 40x speedup when operating in the polystore regime.

## 6.2 Extending PACB$^{\text{opt}}$ to Bag Semantics

We extended PACB$^{\text{opt}}$ to find equivalent rewritings of an input conjunctive query under bag semantics (the original PACB only addresses set semantics). The extension involves a modification to phase (*iv*). In its original form, the matches from $Q$ into the backchase result $B$ are not necessarily injective, being based on *homomorphisms* [6]. These allow multiple atoms from $Q$ to map into the same atom of $B$. To find bag-equivalent rewritings, we disallow such matches, requiring match homomorphisms to be injective.

## 6.3 PACB$^{\text{opt}}_{\text{qbt}}$: Extending PACB$^{\text{opt}}$ to QBTs

The original PACB algorithm (and our extensions thereof) have so far been defined for conjunctive queries only. However, recall that QBTs (Section 4.1) are nested.

[4]Termination of the chase with tgds is undecidable [16], but many sufficient conditions for termination are known, starting with weak acyclicity [22]. Our constraints are chosen so as to ensure termination.

[5]A rewriting of $Q$ is join-minimal if none of its joins can be removed while preserving equivalence to $Q$.

We extend the PACB$^{\text{opt}}$ algorithm to nested QBTs as follows. Each block $B$ is rewritten *in the context of its ancestor blocks* $A_1, \ldots, A_n$, to take into account the fact that the free variables of $B$ are instantiated with the results of the query corresponding to the conjunction of its ancestor blocks. We therefore replace $B$ with the rewriting of the query $A_1 \wedge A_2 \wedge \ldots \wedge A_n \wedge B$. We call the resulting algorithm PACB$^{\text{opt}}_{\text{qbt}}$, using the notation PACB$^{\text{opt}}_{\text{qbt}}\langle C, c\rangle$ to emphasise the fact that it is parameterized by the constraints $C$ and the cost model $c$.

Recalling the uninterpreted (black-box) function semantics from Section 2, Corollary 6.2 implies:

Corollary 6.3. *Under uninterpreted-function semantics, Corollary 6.2 still holds when we replace conjunctive queries with QBT queries and PACB$^{opt}$ with PACB$^{opt}_{qbt}$.*

# 7 GUARANTEES ON THE REDUCTION

We provide the following formal guarantees for our solution to the cross-model rewriting problem based on the reduction to single-model rewriting. Recall from Section 3 that $enc(M)$ are the relational constraints used to encode $M \in \mathcal{M}$ in virtual relations, $enc(Q)$ encodes a $QBT^{XM}$ query $Q$ as a QBT query over the virtual relations, and $dec(R)$ decodes a QBT query over the virtual relations into a $QBT^{XM}$ query. Also recall from Section 6 that given a set $\mathcal{V}$ of $QBT^{XM}$ views, $C_{\mathcal{V}}$ are the relational constraints used to capture $\mathcal{V}$. We have:

Theorem 7.1 (Soundness of the reduction). *Let $Q$ be a $QBT^{XM}$ query over a polystore over the set of data models $\mathcal{M}$. Let $\mathcal{I}$ be a set of integrity constraints satisfied by the data in the polystore and $enc(\mathcal{I})$ be their relational encoding.*

*For every rewriting $R$ obtained via our reduction, i.e.*

$$R = dec(\, PACB^{opt}_{qbt}\langle enc(\mathcal{I}) \cup \bigcup_{M\in\mathcal{M}} enc(M) \cup C_{\mathcal{V}}, c\rangle(enc(Q))\,),$$

*$R$ is a $c$-optimal $\mathcal{V}$-based rewriting equivalent to $Q$ under $\mathcal{I}$, assuming black-box function semantics (Section 2).*

In the above, $enc(\mathcal{I}) \cup \bigcup_{M\in\mathcal{M}} enc(M) \cup C_{\mathcal{V}}$ is the set of constraints used by the PACB$^{\text{opt}}_{\text{qbt}}$ algorithm; Theorem 7.1 states the correction of our approach, outlined in Figure 1.

# 8 EXPERIMENTAL EVALUATION

Below, we describe experiments demonstrating the efficiency and effectiveness of our cross-model rewriting technique.

**Hardware Platform.** We used commodity cluster machines with an Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz, 40Cores, 123GB RAM, disk read speed 616 MB/s, and disk write speed 455 MB/s (interesting since some systems, e.g., AsterixDB, write intermediate results to disk).

**Polystore Configuration.** For our main polystore setting (called "ESTOCADA polystore engine" hereafter) we use [1] as a light-weight execution engine and a set of data stores, selected for their capabilities and popularity: an RDBMS (Postgres v9.6), JSON document stores (Postgres v9.6, MongoDB v4.0.2) and AsterixDB v4.9, SparkSQL v2.3.2 and a text search engine (Solr v6.1). We set a 60GB buffer cache for all systems, we configured the number of utilizable

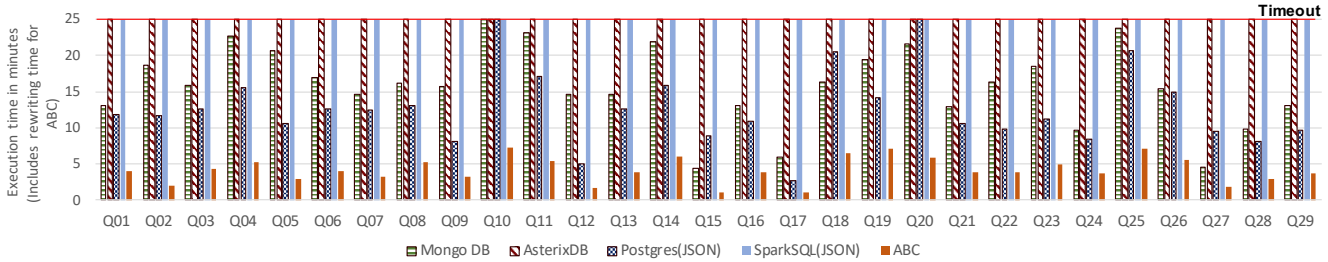


**Figure 6: ESTOCADA evaluation (relational and JSON views in Postgres, JSON view in Solr) vs. single-store evaluation**

cores to 39, and we assigned 8GB to the working memory (the default for Postgres). We set AsterixDB's compiler frame size (the page size for batch query processing) to 6MB.

**Dataset.** We used the real-life **46.6 GB** MIMIC-III dataset [36] described in Section 1.1.

**Generating Query and View Families.** We created a polystore benchmark based on MIMIC as follows. We defined a set $QT$ of 25 query templates, each checking meaningful conditions against a patient's data. These are parameterized by the patient id and by other selection constants, and they involve navigation into the document structure. Each query/view is obtained by joining a subset of $QT$ and picking values for the selection constants, leading to an exponential space of meaningful queries and views. Among them, we identified those with non-empty results: first, non-empty instantiations of $QT$ queries, then joins of two such queries, then three etc., in an adaptation of the Apriori algorithm [9].

**Example** 6. *Consider the query templates $QT_0$, $QT_1$ and $QT_2$ in Appendix H, shown directly in relationally encoded form. $QT_0$ asks for patients' information including patient's PATIENTID, DOB, GENDER and DOD (if available). $QT_1$ asks for all "abnormal" lab measurement results for patients. $QT_2$ asks for bloodwork-related lab measurements. The natural join of $QT_0$, $QT_1$ and $QT_2$ yields a query $Q$ which seeks information on patients with abnormal blood test results.* □

**The Queries.** We chose 50 queries $Q_{EXP}$ among those described above (we show examples in Appendixes D, E, F and G). 58% of the queries ($Q_{01}, \ldots, Q_{29}$) contain full-text operations; all involve joins, selections, and at least two-level-deep navigation into the JSON document structure.

**The Views.** We materialized a set of views $V_{EXP}$ as follows.

We stored in Postgres 6 relational views $V_{PostgreSQL} \subset V_{EXP}$ of the MIMIC-JSON dataset, comprising the uniformly structured part of the dataset (including all patient chart data and admission under specific services such as Cardiac Surgery, Neurologic Medical, etc).

We also stored in Postgres a set $V_{PostgresJSON} \subset V_{EXP}$ of 21 views which are most naturally represented as nested JSON documents. These views store for instance: (*i*) lab tests conducted for each patient with "abnormal" test results (e.g., blood gas, Cerebrospinal Fluid (CSF)); (*ii*) data about drugs prescribed to patients sensitive to certain types of antibiotics (e.g., CEFEPIME); (*iii*) data about drugs and lab test prescribed to each patient who underwent specific types of procedures (e.g.,carotid endarterectomy); (*iv*) microbiology tests conducted for each patient; (*v*) lab tests for each patient who was prescribed certain drug types (e.g. additive, base); etc.

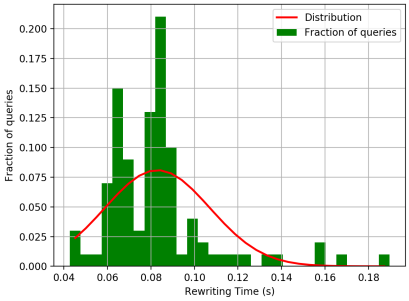


**Figure 8: Rewriting time (28 relevant views)**

We placed in Solr a view $V_{Solr} \in V_{EXP}$, storing for each patient the caregivers' reports (unstructured text). The usage of AsterixDB, SparkSQL and MongoDB is detailed below.

## 8.1 Cross-Store Query Rewritings Evaluation

We study the effectiveness of ESTOCADA cross-store query rewriting: (*i*) compared to single-store query evaluation (Section 8.1.1); (*ii*) improving performance in the pre-existing polystore engines: BigDAWG [21] and ESTOCADA polystore engine (Section 8.1.2).

*8.1.1* ***Single-Store Evaluation Comparison.*** Figure 8 shows the rewriting time for the query set and views described above. Notice that most queries are rewritten within 100ms and the few outliers require at most 190ms, which is negligible compared to the query execution time (in the order of minutes, as seen in Figures 6 and 7). Figure 9 shows the distribution of rewriting time over the same query set when we scale up the number of relevant views (we did not materialize them) to 128 views (this is for stress-test purposes, as 128 views relevant to the data touched by a single query is unlikely).

**Queries with Text Search Predicates.** Figure 6 reports the total execution plus rewriting time of ESTOCADA using the above views for $Q_{01}$ to $Q_{29}$, all of which feature a text search predicate against the text notes in the caregiver's report. For each query, cross-model rewriting and evaluation significantly outperforms direct evaluation in any single store.

For SparkSQL and AsterixDB, queries ($Q_{01}$,…, $Q_{29}$) took over 25 minutes (the timeout value we used). The bottleneck is the text search operation: full-text indexing is lacking in SparkSQL, and limited to exclude text occurring within JSON arrays in AsterixDB. This confirms our thesis that cross-model redundant storage of data into the stores most suited for an expected operation is worthwhile.

For MongoDB and Postgres, the execution time is correlated with the number of JSON array unnest operators. For instance, query $Q_{25}$ has 5 unnest operators whereas query $Q_{17}$ has 2. Postgres outperforms MongoDB because the latter lacks join-reordering

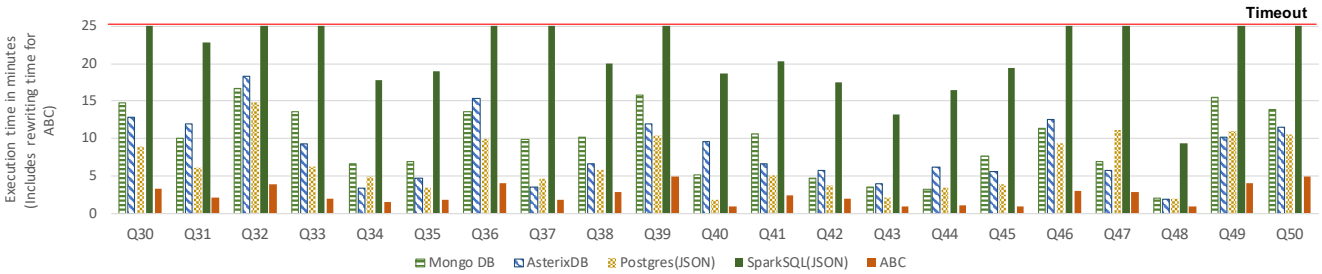


**Figure 7: ESTOCADA query evaluation (relational, resp. JSON views in Postgres) vs. single-store query evaluation**

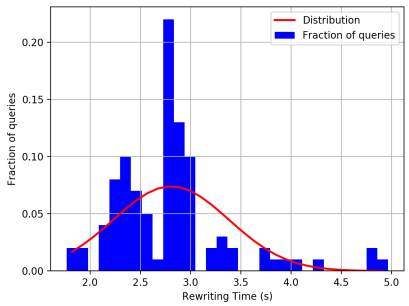


**Figure 9: Rewriting time (128 relevant views)**

optimization, and it does not natively support inner joins. These must be simulated by left outer joins – using the $lookup operator – followed by a selection for non-null values –using the $match operator.

**Queries without Text Search Predicates.** Figure 7 repeats this experiment for queries $Q_{30}$,…,$Q_{50}$, which do not perform text search. These queries each feature join, selection and navigation into the JSON document structure (at least two levels deep). The relevant views for these queries are $V_{PostgreSQL} \cup V_{PostgresJSON}$; again, exploiting them outperforms single-store evaluation. SparkSQL has the highest query execution time; this is because it only supports navigation into JSON arrays through the EXPLODE function, which is highly inefficient (see Appendix F for examples). AsterixDB is outperformed because its optimizer does not support join reordering. In single-store evaluation, Postgres is more efficient; this is why we chose it to store $V_{PostgresJSON}$.

**Materializing All Views In a Single Store.** We performed experiments showing that cross-model evaluation is more efficient than view-based single-store evaluation: on Postgres, for most of our queries; for the other systems, for all our queries. The details can be found in Appendix L.

*8.1.2* ***Polystore Evaluation Comparison.*** We study the benefits that ESTOCADA can bring to an existing polystore system by transparently rewriting queries using materialized views stored across different stores. We used two polystore engines: (*i*) ESTOCADA polystore engine instantiated with two Postgres servers (one stores relational data while the other stores JSON) and Solr v6.1 for storing text; (*ii*) the latest release of the BigDAWG polystore. A key BigDAWG concept is an *island*, or collection of data stores accessed with a single query language. BigDAWG supports islands for relational, array and text data, based on the Postgres, SciDB [5] and Apache Accumulo [2] stores, respectively. BigDAWG queries use explicit CAST operators to *migrate* an intermediary result from an island to another. To work with BigDAWG, we extended it with two new CAST operators: (*i*) to migrate Solr query results to Postgres; (*ii*) to migrate Postgres JSON query results to a Postgres relational instance and vice-versa. The main difference between our ESTOCADA polystore engine and BigDAWG is that we join subquery results *in the mediator* using a BindJoin[51], whereas BigDAWG *migrates* such results to an island capable of performing the join (typically, a relational one).

**Data Storage** We store the MIMIC-III dataset (in both systems) as follows: patient metadata in Postgres relational instance; caregivers' reports in Solr; patients' lab events, prescriptions, microbiology events, and procedures information in the Postgres JSON instance.

**Polystore Queries.** We rewrote $Q_{01}$,…, $Q_{29}$ in BigDAWG syntax, referring to each part of the data from its respective island (as BigDAWG requires); we refer to the resulting query set as $Q_{BigDAWG}$. Appendix J illustrates a BigDAWG query. We have the same set of queries in $QBT^{XM}$ syntax; we call these queries $Q_{\text{ESTOCADA } Polystore}$.

**The Views.** To the view set $V_{EXP}$ introduced above, we have added a new set of views $V_{NEW}$ which can benefit $Q_{BigDAWG}$ and $Q_{\text{ESTOCADA } Polystore}$ queries as we detail below.

The experiment queries vary in terms of the full-text search predicates selectivity. 60% of $Q_{BigDAWG}$ and $Q_{\text{ESTOCADA } Polystore}$ queries consist of full-text search predicates, which are *not highly selective* (e.g., "low blood pressure"). We refer to these queries as $Q^{60\%}_{BigDAWG}$ and $Q^{60\%}_{\text{ESTOCADA } Polystore}$. We observed that the cost of such queries in BigDAWG is dominated by moving and ingesting the results of Solr queries in Postgres (relational instance) to join them with the results of other sub-queries. To alleviate this overhead, we have materialized *in the Postgres server storing JSON* a set of 6 views $V_{NEW}$, *which join the data from Solr (using those full-text predicates in the views definitions) with JSON data from the Postgres JSON server.* Given the $Q^{60\%}_{BigDAWG}$ queries, $V_{PostgresJSON}$, $V_{PostgresSQL}$ and $V_{NEW}$, our cross-model views-based rewriting approach finds rewritings using views from Postgres (relational instance) and Postgres (JSON instance). The performance saving is due to the fact that we no longer have to move data from Solr to a relational Postgres instance (see Figure 10 for queries labeled *). Although ESTOCADA polystore engine does not require any data movement, it still benefits from utilizing $V_{PostgresJSON}$, $V_{PostgresSQL}$ and $V_{NEW}$ to answer $Q^{60\%}_{\text{ESTOCADA } Polystore}$ queries as shown in Figure 10 (queries labeled with *).

In contrast, the remaining 40% of $Q_{BigDAWG}$ and $Q_{\text{ESTOCADA } Polystore}$ queries have *highly selective full-text search predicates* (we refer to these as queries $Q^{40\%}_{BigDAWG}$ and $Q^{40\%}_{\text{ESTOCADA } Polystore}$). The high

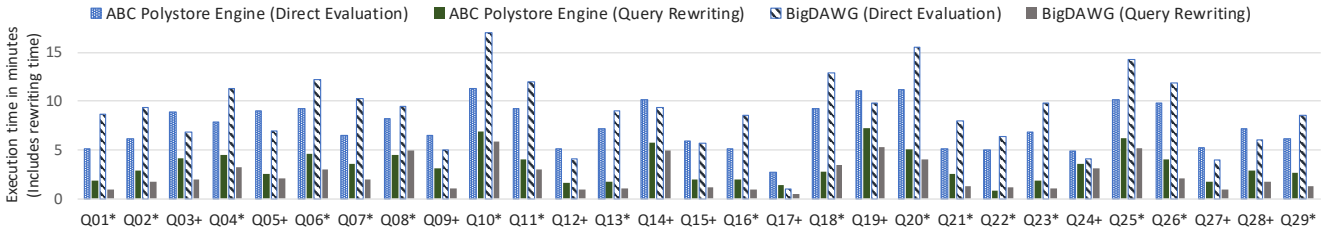


**Figure 10: Queries and Rewritings Evaluation In Polystore Engines**

selectivity of these queries reduces the overhead of moving the data from Solr to Postgres (relational instance) in BigDAWG, and in general the data movement is not a bottleneck for these queries. However, both systems can benefit from the materialized views $V_{EXP}$ to evaluate these queries as shown in Figure 10 (queries labeled with +).

As mentioned earlier, ESTOCADA polystore engine and BigDAWG differ in terms of multi-store join evaluation strategies, leading to different performance variations when the queries and/or views change. On the queries $Q^{60\%}_{BigDAWG}$ and $Q^{60\%}_{\text{ESTOCADA } Polystore}$, where the full-text search predicates are not very selective, BigDAWG execution time is dominated by moving partial results to the join server. In contrast, ESTOCADA polystore engine performs better since it computes the join in memory in the mediator, thus it does not need to pay the intermediary result storage cost. For the other 40% of queries (with very selective full-text search predicates), BigDAWG data movement cost is negligible, thus its evaluation time is dominated by evaluating the join between sub-queries results in the Postgres relational island, where join algorithms and orders may be better chosen than in the ESTOCADA polystore engine (the two platform support different sets of join algorithms). However, the differences are negligible. The main conclusion of this experiment, however, is that our cross model views-based rewriting approach *improves performance in both polystore engines.*

## 8.2 PACB VS PACB$^{opt}$

This experiment demonstrates the performance gains of PACB$^{opt}$ over PACB in a polystore setting. We consider the queries $Q_{EXP}$ and the 28 views $V_{EXP}$ that can be utilized to answer $Q_{EXP}$, introduced above. We add to $V_{EXP}$ some *irrelevant* views $V_{irrel} \subseteq (\mathcal{V} - V_{EXP})$. We scale the size of $V_{irrel}$ from 1000 to 4000. Figure 11 presents the *average rewriting time* of $Q_{EXP}$ in the presence of $V_{EXP} \cup V_{irrel}$; the $y$ axis is in log scale.

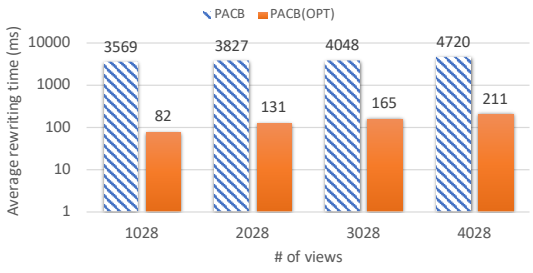


**Figure 11: PACB vs PACB$^{opt}$ rewriting performance**

## 8.3 Summary of Experimental Findings

We have shown that our cross-model views-based rewriting approach is *portable* across polystore engines. Moreover, it is *worthwhile*, as it improves their performance in natural scenarios for both cross-model and even single-model user queries; the latter are outperformed by rewritings using a distributed cross-store (and cross-model) set of materialized views (even when accounting for the time it takes to find the rewriting). We have also shown that the time spent searching for rewritings is a small fraction of the query execution time and hence a worthwhile investment. As we confirm experimentally, the performance of the rewriting search is due to our optimized PACB$^{opt}$ algorithm, shown to outperform standard PACB by up to 40×.

# 9 RELATED WORK

Heterogeneous data integration is an old topic [29, 31, 42, 44] addressed mostly in a single-model (typically relational) setting, where cross-model query rewriting and execution did not occur. The federated Garlic system [29] considered true multi-model settings, but the non-relational stores did not support their own declarative query language (they included for instance text documents, spreadsheets, etc.), being instead wrapped to provide an SQL API. Consequently works on federated databases did not face cross-model rewriting problem.

The remark "one-size does not fit all" [54] has been recently revisited [35, 43] for heterogeneous stores. [40] uses a relational database as a "cache" for partial results of a MapReduce computation, while [41] considers view-based rewriting in a MapReduce setting. Unlike our work, these algorithms need the data, views and query to be in the same data model.

Polystores [7, 20, 21] allow querying heterogeneous stores by grouping similar-model platform into "islands" and explicitly sending queries to one store or another; data sets can also be migrated by the users. Our LAV approach is novel and, as we have shown, enables to improve the performance of such stores also. The integration of "NoSQL" stores has been considered e.g., in [12] again in a GAV approach, without the benefits of LAV view-based rewriting.

Adaptive stores for a single data model have been studied e.g., in [10, 14, 32, 37, 38]; views have been also used in [8, 53] to improve the performance of a large-scale distributed relational store. The novelty of ESTOCADA here is to support multiple data models, by relying on powerful query reformulation techniques under constraints.

Data exchange tools such as Clio [22, 30] allow migrating data between two different schemas of the same (typically relational and sometimes XML) model (and thus not focused on cross-model rewriting). We aim at providing to the applications transparent data access to heterogeneous systems. View-based rewriting and view selection are grounded in the seminal works [31, 42]; the latter focuses on maximally contained rewritings, while we target exact query rewriting, which leads to very different algorithms.

Further setting our work apart is the scale and usage of integrity constraints. Our pivot model recalls the ones described in [18, 44] but ESTOCADA generalizes these works by allowing multiple data models both at the application and storage level.

CloudMdsQL [39] is an integration language resembling $QBT^{XM}$, and our cross-model view-based rewriting approach could be easily adapted to use CloudMdsQL as its integration language, just like we adapted it to use BigDAWG's. The polystore engine supporting CloudMdsQL does not feature our cross-model view-based query rewriting functionality.

Works on publishing relational data as XML [24] followed the GAV paradigm, thus did not raise the (cross-model) view-based rewriting problem we address here.

## 10 CONCLUSIONS

We have shown that hybrid (multi-store) architectures have the potential to significantly speed up query evaluation by materializing views in the systems most suited to expected workload operations, even when these views are distributed across stores and data models. ESTOCADA supports this functionality by a local-as-view approach whose immediate benefit is flexibility since it requires no work when the underlying data storage changes. To make this approach feasible, we had to couple modeling contributions (in designing a reduction from multi-model view-based rewriting to relational rewriting under constraints), with algorithmic contributions (in optimizing and extending the state-of-the-art algorithm for relational view-based rewriting under constraints) and with systems contributions (integrating our rewriting algorithm within [1] and BigDAWG).

In our experiments, we achieved performance gains by simply placing the materialized views according to a few common-sense guidelines (e.g., place large unstructured text collections in a store with good full-text indexing support, and place inherently nested data in JSON document stores instead of normalizing it into flat relational tables). As a natural future work step, we are targeting the cost-based recommendation of optimal cross-model view placement.

## REFERENCES


[1] [n. d.]. Anonymized for double-blind purposes.
[2] [n. d.]. Apache Accumulo.https://accumulo.apache.org/.
[3] [n. d.]. AsterixDB. https://asterixdb.apache.org/.
[4] [n. d.]. Saxon.http://saxon.sourceforge.net/.
[5] [n. d.]. Scidb.https://www.paradigm4.com/.
[6] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.
[7] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed K. Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, Saravanan Thirumuruganathan, and Anis Troudi. 2018. RHEEM: Enabling Cross-Platform Data Processing - May The Big Data Be With You! *PVLDB* (2018).
[8] Parag Agrawal, Adam Silberstein, Brian F. Cooper, Utkarsh Srivastava, and Raghu Ramakrishnan. 2009. Asynchronous view maintenance for VLSD databases. In *SIGMOD*.
[9] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *PVLDB*.
[10] I. Alagiannis, S. Idreos, and A. Ailamaki. 2014. H2O: a hands-free adaptive store. In *SIGMOD*.
[11] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark SQL: Relational data processing in Spark. In *SIGMOD*. ACM.
[12] P. Atzeni, F. Bugiotti, and L. Rossi. 2014. Uniform access to NoSQL systems. *Information Systems* (2014).
[13] Ding Chen and Chee-Yong Chan. 2010. ViewJoin: Efficient view-based evaluation of tree pattern queries. In *ICDE*.
[14] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *PVLDB* (2011).
[15] Alin Deutsch, Bertram Ludäscher, and Alan Nash. 2005. Rewriting Queries Using Views with Access Patterns Under Integrity Constraints. In *ICDT*.
[16] Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. 2008. The chase revisited. In *PODS*.
[17] Alin Deutsch, Lucian Popa, and Val Tannen. 2006. Query reformulation with constraints. *SIGMOD* (2006).
[18] Alin Deutsch and Val Tannen. 2003. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *In VLDB*.
[19] Alin Deutsch and Val Tannen. 2003. Reformulation of XML Queries and Constraints. In *ICDT*.
[20] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik. 2015. The BigDAWG Polystore System. *SIGMOD* (2015).
[21] A. Elmore, J. Duggan, M. Stonebraker, and al. 2015. A Demonstration of the BigDAWG Polystore System. *PVLDB* (2015).
[22] R. Fagin, P. Kolaitis, R. Miller, and L Popa. 2003. Data Exchange: Semantics and Query Answering. In *ICDT*.
[23] Ronald Fagin, Phokion G Kolaitis, Renée J Miller, and Lucian Popa. 2005. Data exchange: semantics and query answering. *Theoretical Computer Science* (2005).
[24] Mary Fernández, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, and Wang-Chiew Tan. 2002. SilkRoute: A framework for publishing relational data in XML. *TODS* (2002).
[25] Daniela Florescu, Alon Y. Levy, Ioana Manolescu, and Dan Suciu. 1999. Query Optimization in the Presence of Limited Access Patterns. In *SIGMOD*.
[26] Georges Gardarin, Fei Sha, and Zhao-Hui Tang. 1996. Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System. In *VLDB*. 378–389. http://www.vldb.org/conf/1996/P378.PDF
[27] Gösta Grahne and Alberto O. Mendelzon. 1999. Tableau Techniques for Querying Information Sources through Global Schemas. In *ICDT*.
[28] Gosta Grahne and Adrian Onet. 2013. Anatomy of the chase. *arXiv preprint arXiv:1303.6682* (2013).
[29] Laura Haas, Donald Kossmann, Edward Wimmers, and Jun Yang. 1997. Optimizing queries across diverse data sources. (1997).
[30] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. 2005. Clio grows up: from research prototype to industrial tool. In *SIGMOD*.
[31] Alon Y Halevy. 2001. Answering Queries Using Views: A Survey. *The VLDB Journal* (2001).
[32] S. Idreos, M.L. Kersten, and S. Manegold. 2007. Database Cracking. In *CIDR*.
[33] Ioana Ileana, Bogdan Cautis, Alin Deutsch, and Yannis Katsis. 2014. Complete yet practical search for minimal query reformulations under constraints. In *SIGMOD*.
[34] ISO/IEC 9075-1:1999. [n. d.]. SQL:1999. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=26196. Accessed March 2016.
[35] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. 2013. WWHow! Freeing Data Storage from Cages. In *CIDR*.
[36] AEW Johnson, J.J. Pollard, L. Shen, L. Lehman, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L.A. Celi, and R.G. Mark. 2016. MIMIC-III, a freely accessible critical care database. *Scientific Data (2016). DOI: 10.1038/sdata.2016.35. Available at: http://www.nature.com/articles/sdata201635* (2016).
[37] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. 2015. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *CIDR*.
[38] Asterios Katsifodimos, Ioana Manolescu, and Vasilis Vassalos. 2012. Materialized view selection for XQuery workloads. In *SIGMOD*.
[39] Boyan Kolev, Patrick Valduriez, Carlyna Bondiombouy, Ricardo Jiménez-Peris, Raquel Pau, and José Pereira. 2016. CloudMdsQL: querying heterogeneous cloud data stores with a common language. *Distributed and parallel databases* (2016).
[40] J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, and al. 2014. MISO: souping up big data query processing with a multistore system. In *SIGMOD*.
[41] Jeff LeFevre, Jagan Sankaranarayanan, Hakan Hacigümüs, and al. 2014. Opportunistic physical design for big data analytics. In *SIGMOD*.
[42] A. Levy, A. Rajaraman, and J. Ordille. 1996. Querying Heterogeneous Information Sources Using Source Descriptions. In *VLDB*.
[43] H. Lim, Y. Han, and S. Babu. 2013. How to Fit when No One Size Fits.. In *CIDR*.
[44] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. 2001. Answering XML Queries on Heterogeneous Data Sources. In *VLDB*.
[45] Alan Nash and Bertram Ludäscher. 2004. Processing First-Order Queries under Limited Access Patterns. In *PODS*.
[46] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. 2006. Rewriting Nested XML Queries Using Nested Views. In *SIGMOD*.
[47] M. T. Özsu and P. Valduriez. 2011. *Principles of Distributed Database Systems, Third Edition.* Springer.
[48] Yannis Papakonstantinou. 2016. Semistructured Models, Queries and Algebras in the Big Data Era: Tutorial Summary. In *SIGMOD*.

[49] Derek Phillips, Ning Zhang, Ihab F. Ilyas, and M. Tamer Özsu. 2006. InterJoin: Exploiting Indexes and Materialized Views in XPath Evaluation. In *SSDBM*.
[50] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. 2000. A chase too far?. In *ACM SIGMOD Record*, Vol. 29. ACM, 273–284.
[51] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. 1995. Answering Queries Using Templates with Binding Patterns. In *PODS*.
[52] P. Griffith Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD*.
[53] J. Shute, R. Vingralek, B. Samwel, and al. 2013. F1: A Distributed SQL Database That Scales. In *PVLDB*.
[54] M. Stonebraker and U. Cetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*.

# A RELATIONAL ENCODING OF MOTIVATING $QBT^{XM}$ QUERY $Q_1$

```
Q1<patientID, drug, admissionLoc, admissionTime>:-
MIMIC(M),
Child_J(M, patientID, "patientID", "o"),
Child_J(M,A, "admissions", "o"),
Child_J(A, admissionID, "admissionID", "o"),
Child_J(A, report, "reports", "o"),
    Value_J(report, "like-coronary artery"),
Child_J(A, admissionLoc, "admissionLoc", "o"),
Child_J(A, admissionTime, "admissionTime", "o"),
Child_J(A, LE, "labevents", "o"),
Child_J(LE, flag, "flag", "o"),
    Value_J(flag, "abnormal"),
Child_J(LE, labItemID, "id", "o"),
Child_J(A, P, "prescriptions", "o"),
Child_J(P, drug, "drug", "o"),
Child_J(P, drugtype, "drugtype", "o"),
    Value_J(drugtype, "additive"),
LABITEMS(L),
Child_J(L, itemID, "itemID", "o"),
Child_J(L, category, "category", "o"),
    Value_J(category, "blood"),
Eq_J(itemID, labItemID)
```

# B VIEWS CONSTRAINTS

```
V1 Forward (IO) Constraints
MIMIC(M),
Child_J(M, patientID, "patientID", "o"),
Child_J(M,A, "admissions", "o"),
Child_J(A, admissionID, "admissionID", "o"),
Child_J(A, admissionID, "admissionID", "o"),
Child_J(A, report, "reports", "o") ->
V1(d1),
Child_JV1(d1, patientID, "patientID", "o"),
Child_JV1(d1, admissionID, "admissionID", "o"),
Child_JV1(d1, report, "report", "o")
V1's BACKWARD (OI) CONSTRAINTS:
V1(d1),
Child_JV1(d1, patientID, "patientID", "o"),
Child_JV1(d1, admissionID, "admissionID", "o"),
Child_JV1(d1, report, "report", "o")->
MIMIC(M),
Child_J(M, patientID, "patientID", "o"),
Child_J(M,A, "admissions", "o"),
Child_J(A, admissionID, "admissionID", "o"),
Child_J(A, admissionID, "admissionID", "o"),
Child_J(A, report, "reports", "o")
V2's BACKWARD (OI) CONSTRAINTS:
V2(patientID, admissionID, admissionLoc, admissionTime)->
MIMIC(M),
Child_J(M, patientID, "patientID", "o"),
Child_J(M,A, "admissions", "o"),
Child_J(A, admissionID, "admissionID", "o"),
Child_J(A, admissionLoc, "admissionLoc", "o"),
Child_J(A, admissionTime, "admissionTime", "o"),
Child_J(A, report, "reports", "o")
```

# C DECODING OF REWRITING $RWQ_1$

```
FOR SJ:{q=report:coronary artery&
        f1=patientID, admissionID},
    PR:{SELECT patientID AS PID1,
               admissionID AS AID1,
               admissionLoc AS admissionLoc,
               admissionTime AS admissionTime
        FROM   V2},
    PJ:{SELECT v3->'patientID' AS PID2,
               A->'admissionID' AS AID2,
               D->'drug' AS drug
        FROM   V3 v3,
               jsonb_array_elements(v3->'admission') A,
               jsonb_array_elements(A->'drugs') D
        WHERE  D->>'drugtype'='additive'}
WHERE patientID=PID1 AND
      admissionID=AID1 AND
      PID1= PID2 AND
      AID1=AID2
RETURN AJ:{ "patientID":patientID,
            "admissionLoc":admissionLoc,
            "admissionTime":admissionTime,
            "drug":drug}
```

# D SAMPLE JSON QUERY IN POSTGRES SYNTAX

```
Q01:
SELECT  D.PATIENT->>'PATIENTID', LI->>'LABEL'
FROM    MIMIC AS D, LABITEMS AS LI,
        jsonb_array_elements(D.PATIENT->'ADMISSIONS') AS A,
        jsonb_array_elements(A->'LABEVENTS') AS LE,
        jsonb_array_elements(A->'NOTEEVENTS') AS NE
WHERE   LI.LABITEM->'ITEMID'=LE->'ITEMID' AND
        LI.LABITEM@>'{"FLUID":"Blood"}' AND
        LI.LABITEM@>'{"CATEGORY":"Blood Gas"}' AND
        LE->>'FLAG'='abnormal' AND
        to_tsvector('english', NE.text::text) @@
        plainto_tsquery('english','respiratory failure')
```

# E SAMPLE ASTERIXDB QUERY (SQL++)

```
Q01:
USE MIMICiii;
SELECT  D.PATIENTID, LI.LABEL
FROM    MIMIC AS D, LABITEMS LI,
        D.ADMISSIONS AS A, LABEVENTS AS LE,
        A.NOTEEVENTS AS N
WHERE   LI.ITEMID=LE.ITEMID AND
        LE.FLAG="abnormal" AND
        LI.CATEGORY="Blood Gas" AND
        LI.FLUID="Blood" AND
        contains(N.text,"respiratory failure")
```

# F SAMPLE SPARKSQL QUERY

```
Q01:
SELECT  D.PATIENTID, LI.LABEL
FROM    LABITEMS AS LI, MIMIC AS M
        LATERAL VIEW explode(ADMISSIONS) AS A
        LATERAL VIEW explode(A.LABEVENTS) AS LE
        LATERAL VIEW explode(A.NOTEEVENTS) AS N
WHERE   LI.ITEMID=LE.ITEMID AND
        LE.FLAG="abnormal" AND
        LI.CATEGORY="Blood Gas" AND
        LI.FLUID="Blood" AND
        N.text LIKE '%respiratory failure%'
```

# G SAMPLE MONGODB QUERY

```
Q02:
{
  "aggregate": "mimic",
  "pipeline": [
    { "$match": { "$text": {"$search": "\"respiratory failure\"" }}},
    { "$unwind": "$ADMISSIONS" },
    { "$unwind": "$ADMISSIONS.LABEVENTS" },
    { "$lookup": {
        "from": "d_labitems",
        "let": { "item": "$ADMISSIONS.LABEVENTS.ITEMID" },
        "pipeline": [
          { "$match": { "$expr": { "$and":
               [ { "$eq": [ "$ITEMID", "$$item" ]},
                 { "$eq": [ "$FLUID", "Blood" ] } ] } } },
          { "$project": { "ITEMID": 0, "_id": 0 } }
        ], "as": "t"
    }
    },
    { "$match": { "t": { "$ne": [] } } },
    { "$unwind": "$ADMISSIONS.ICUSTAYS"},
    { "$unwind": "$ADMISSIONS.ICUSTAYS.PRESCRIPTIONS"},
    { "$match": {
        "$expr": {
          "$and": [
            { "$eq": [
```

```
                "$ADMISSIONS.ICUSTAYS.PRESCRIPTIONS.DRUG_TYPE",
                "ADDITIVE" ] },
            { "$eq": [
                "$ADMISSIONS.ICUSTAYS.PRESCRIPTIONS.DRUG",
                "Potassium Chloride" ] },
            { "$eq": [
                "$ADMISSIONS.LABEVENTS.FLAG",
                "abnormal" ] } ] } } },
    { "$project": { "t.LABEL": 1, "t.CATEGORY": 1} } ]
}
```

# H QUERY TEMPLATES $QT_0$, $QT_1$, AND $QT_2$

```
QT0<PATIENTID,DOB,DOD,GENDER> :-
MIMIC(M),
ChildJ(M,PATIENTID,"PATIENTID","o"),
ChildJ(M,DOB,"DOB","o"),
ChildJ(M,DOD,"DOD","o"),
ChildJ(M,GENDER,"GENDER","o");
QT1<PATIENTID,ITEMID,VALUE,CHARTTIME> :-
MIMIC(M),
ChildJ(M,PATIENTID,"PATIENTID","o"),
ChildJ(M,A,"ADMISSIONS","o"),
ChildJ(A,LE,"LABEVENTS","o"),
ChildJ(LE,ITEMID, "ITEMID","o"),
ChildJ(LE,VALUE, "VALUE","o"),
ChildJ(LE,CHARTTIME, "CHARTTIME","o"),
ChildJ(LE,FLAG, "FLAG","o"),
    ValueJ(FLAG,"abnormal")
QT2<ITEMID,LABEL,FLUID>:-
LABITEMS(L),
ChildJ(L, ITEMID, "ITEMID","o"),
ChildJ(L,CATEGORY, "CATEGORY","o"),
ChildJ(L,LABEL, "LABEL","o"),
ChildJ(L,FLUID, "FLUID","o"),
    ValueJ(FLUID,"Blood")
```

# I SYNTAX OF $QBT^{XM}$ QUERY LANGUAGE

```
QBTXMBlock -> forPattern wherePattern? returnPattern
forPattern -> FOR pattern (',' pattern)*
wherePattern -> WHERE condition
returnPattern -> RETURN constructor
pattern -> annotation ':' '{' modelPattern '}'
modelPattern -> AJPattern
        | PJPattern
        | RKPattern
condition -> term | term=term | '(' condition ')'
          | condition AND condition
term -> constant | variable
     | annotation ':' '{' modelPathExpr '}'
     | funCall
funCall -> funName '(' args? ')'
args -> term (',' term)*
modelPathExpr -> AJPathExpr | PJPathExpr | RKLookUpExpr

constructor -> annotation ':' '{' modelConstructor '}'
modelConstructor -> AJObjConstructor
           | PJObjConstructor
           | RKMapConstructor

//* AJ Pattern *//
AJPattern -> AJForPattern AJWherePattern?
AJForPattern -> variable IN collectionName
        | variable IN AJPathExpr
AJWherePattern -> WHERE AJCondition
AJCondition -> AJterm | AJterm=AJterm | '(' AJCondition ')'
              | AJCondition AND AJCondition
AJterm -> AJPathExpr | variable | constant | AJFunCall

//* PJ Pattern *//
PJPattern -> PJFromPattern PJWherePattern?
PJFromPattern -> relationName AS variable
              | JSON_ARRAY_ELEMENTS(PJPathExpr) AS variable
PJWherePattern -> WHERE PJCondition
PJCondition -> PJterm
              | PJterm=PJterm
              | '(' PJCondition ')'
              | PJCondition AND PJCondition
PJterm -> PJPathExpr | variable | constant | PJFunCall

//* RK Pattern *//
RKPattern -> variable IN RKLookUpExpr
//* AJ Path Expression *//
AJPathExpr -> variable'.'key ('.'key)*
//* PJ Path Expression *//
PJPathExpr -> variable'->'key('->'key)*
//* RK Look Up Expression *//
RKLookUpExpr -> mainMapName '[' key ']'
              | variable '[' key ']'
//* AJ Constructor *//
AJObjConstructor ->
                  '{' ( key ':' AJvalue(',' key ':' AJvalue )* '}'
AJvalue -> variable | constant | AJPathExpr | AJPattern
         | AJObjConstructor
//* PJ Constructor *//
PJObjConstructor ->
       JSON_BUILD_OBJECT( key,PJvalue (',' key,PJvalue)* )
PJvalue -> variable | constant | PJPathExpr | PJPattern
         | PJObjConstructor
//* RK Constructor */
RKMapConstructor ->
           key ->'{'key ':' variable(',' key ':' variable )*'}'
annotation -> AJ | PR | PJ | SJ | RK
key -> STRING
collectionName -> STRING
mainMapName -> STRING
relationName -> STRING
funName -> STRING
constant -> STRING | INTEGER
```

# J SAMPLE BIGDAWG QUERY

```
Query01:
bdrel(SELECT P.PATIENTID, TAB2.LABEL
      FROM
      bdcast
       (bdtext(q=text:respiratory+failure&fl=PATIENTID,ADMISSIONID),
            TAB1, '(PATIENTID INTEGER, ADMISSIONID INTEGER)',
              relational),
      bdcast
      (bdjq(
        SELECT P.PATIENTID AS PATIENTID,
               (A->>'ADMISSIONID')::int AS ADMISSIONID,
               LE->>'LABEL' AS LABEL
        FROM   PATIENTEVENTS AS P,  LABITEMS LI,
               jsonb_array_elements(P.PATIENTEVENT->'ADMISSIONS') AS A,
               jsonb_array_elements(A->'LABEVENTS') AS LE
        WHERE  LI.LABITEM->'ITEMID'=LE->'ITEMID' AND
               LI.LABITEM@>'{"FLUID":"Blood"}' AND
               LI.LABITEM@>'{"CATEGORY":"Blood Gas"}'),
           TAB2, '(PATIENTID INTEGER, ADMISSIONID INTEGER, LABEL VARCHAR)',
             relational),
      PATIENTS P
      WHERE P.PATIENTID=tab1.PATIENTID AND P.PATIENTID=tab2.PATIENTID)
```

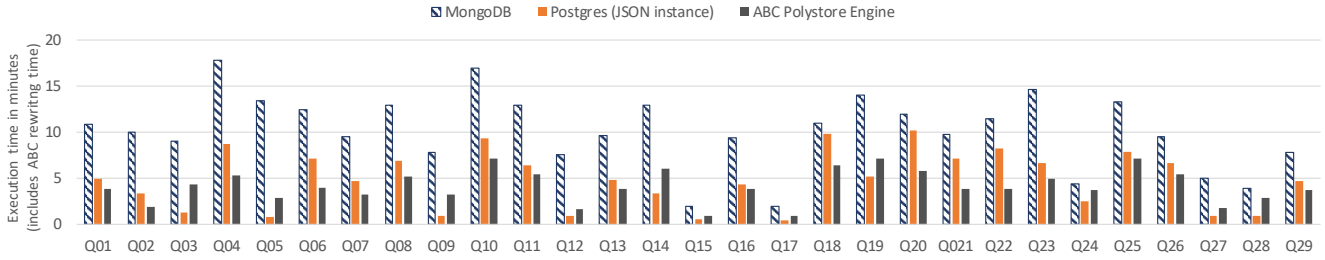


**Figure 12: ESTOCADA cross-store query evaluation (relational, resp. JSON views in Postgres, JSON view in Solr) vs. single-store query evaluation with views (in MongoDB and Postgres).**

## K COST-BASED OPTIMIZATION

Optimization in the ESTOCADA polystore engine used in our experiment combines heuristics and costs, as follows.

First, for each view $V_i$ used in a rewriting $RW$, such that $V_i$ resides in the datastore $DS_i$, $RW$ decoding (Section 5.4) consolidates the maximum number of atoms that can be pushed (evaluated) into $DS_i$, depending on the query capabilities of the latter; we call such an operator *SourceAccess*, and systematically choose to evaluate it in $DS_i$.

The *cost* of *SourceAccess* is estimated using a linear cost model with a fixed start-up component and one that grows linearly with the estimated size of the returned data (see below); this model also uses system (physical) parameters which we computed by running by hand a few "system profiling" queries, notably the observed disk read and write rate (for cold reads) and time to manipulate data in memory for subsequent, hot reads. We had derived these physical parameters for prior work on the same team cluster, thus we just reused those numbers. In general, *calibration* [26] can be used, i.e., a module of the polystore systems runs some controlled experiments on each hardware on which it is newly deployed, and from these experiments derives the values characterizing the system performance. Calibration is present in many modern systems, e.g., RHEEM[6].

The *cardinality* of a *SourceAccess* is estimated using: (*i*) statistics on the views (e.g., number of tuples, or JSON trees, their average size, etc.); these are gathered at view materialization time, together with the minimum, maximum, and number of distinct values for fine-granularity attributes; (*ii*) default constant factors for inequalities or keyword search. Statistic estimates are rarely exact and ours could be perfected; however, they have already helped us make some good choices.

Optimization develops left-deep join plans in a bottom-up fashion based on the *SourceAccess* operators. To place each join, we consider: (*i*) each data source hosting one input, if it supports joins; (*ii*) always, as a fall-back, the mediator. On a full join tree, we push the remaining selection ($\sigma$) and projections ($\pi$) as far as possible (taking into account source capabilities), then add possible (linear-cost) Construct operators which structure the results in the desired data model. The cost of $\sigma$ and $\pi$ are linear in the size of the input with system-dependent constants; the constant for full-text search is higher than for equality selections.

This model allows to find the cheapest plan for $RW$; for a given rewritten query $Q$, the cheapest rewriting is selected.

## L MATERIALIZING ALL VIEWS IN A SINGLE STORE EXPERIMENT

We now compare cross-model evaluation against view-based single-store evaluation. To that purpose, we stored *all* $V_{EXP}$ views in *each* of MongoDB, AsterixDB, SparkSQL and Postgres. As shown in Figure 12, multi-store evaluation outperforms single-store evaluation with views in MongoDB. We note that: (*i*) each MongoDB query joins three materialized views; the required use of the $unwind (a.k.a. unnest) operator inside $lookup operators makes the latter slower. For each document/tuple from an outer collection in the $lookup operator, $unwind unnests the entire inner collection to perform the join, which is inefficient; (*ii*) MongoDB requires to place the full-text search predicate at the beginning of a query pipeline, which can hurt performance if the predicate is not highly selective. In AsterixDB and SparkSQL, *all queries timed out* due to the lack of full-text indexing on collection fields. To try to help them, we flattened the nested documents and created the full text-search index. In this setting, AsterixDB failed with an internal error (ArrayIndexOutOfBoundsException is thrown), while SparkSQL still timed out since it has no full-text index support. For 40% of the queries, single-store view-based evaluation in Postgres outperforms cross-model evaluation, which is the most efficient for 60% of the queries. Note that *Postgres benefits from the view-based rewriting algorithm of ESTOCADA here*. On top of view-based performance improvements, Postgres exploits more choices of join order and algorithms.

[6]https://github.com/rheem-ecosystem/rheem/tree/master/rheem-core/src/main/java/org/qcri/rheem/core/profiling