Toward Scalable Hybrid Stores

Francesca Bugiotti, Damian Bursztyn, Alin Deutsch, Ioana Ileana, Ioana Manolescu

Toward Scalable Hybrid Stores*

Francesca Bugiotti[1,2], Damian Bursztyn[2], Alin Deutsch[3], Ioana Ileana[3], and Ioana Manolescu[2]

[1] CentraleSupélec [2]INRIA & U. Paris-Sud, France [3]UC San Diego, USA

Abstract. Data centric applications often use heterogeneous datasets: some very large while others of moderate size, some highly structured (e.g., relations) while others complex structured (e.g., graphs) or little structured (e.g., log data). Facing them is a variety of storage systems but none of which is the best for all, at all times.
We present ESTOCADA, an architecture we are currently developing to efficiently handle highly heterogeneous datasets based on a dynamic set of potentially very different data stores. ESTOCADA provides to the application layer access to each dataset in its native format, while hosting them internally in a set of potentially overlapping fragments, possibly distributed across heterogeneous stores. At the core of ESTOCADA lie powerful view-based rewriting and view selection algorithms to marry correctness with high performance.

1 Context and outline

Digital data has become central to daily life in modern societies. Data is being produced and consumed in many data models, some of which may be structured (flat and nested relations, tree models such as JSON, graphs such as those encoding RDF data or social networks) and some of which may be less so (e.g., CSV or flat text files). Each of the data types above arises in application scenarios including traditional data warehousing, e-commerce, social network data analysis, Semantic Web data management, data analytics, etc.

It is increasingly the case that an application's needs can no longer be met within a single dataset or even within a single data model. Consider for instance a traditional customer relationship management (CRM) application. While typically CRM needed to deal only with a relational data warehouse, now the application needs to incorporate new data sources in order to build a better knowledge of its customers: (*i*) information gleaned from social network *graphs* about clients' activity and interests, and (*ii*) *log file* from multiple e-commerce stores, characterizing the clients' purchase activity in those stores. Monetizing access to operational databases is predicted to grow[1], thus access to such third-party data sources is increasing.

The in-house RDBMS performs fine on the relational data. However, the social graph data fits badly in that system, and the company attempts to store it

* This paper is a short version of [3].

[1] Gartner predicts that 30% of business will do it by 2016: http://www.gartner.com/newsroom/id/2299315.

in a dedicated graph store, until an engineer argues that it should be decomposed and stored into a highly-efficient NoSQL key-value store system she has just experimented with. The storage and processing of log files is delegated to a Hive installation (over Hadoop), until the summer research intern observes that recent work [13] has shown that *some* data from Hive should be lifted at runtime in the relational data warehouse to gain a few orders of magnitude of performance! Finally an engineer moves part of the log data in the in-memory column store, as well as part of the social data to make their joint exploitation faster.

Deploying and exploiting the CRM application for best performance is set to be a nightmare now. There is little consensus on what systems to use, if any; three successive engineers have recommended (and moved the social data into and out of) three different stores, one for graphs, one for key-value pairs, and the last an in-memory column database. The application is sometimes very slow. Migrating data is painful at every change of system; they are not sure the complete dataset survived at each step, and data keeps accumulating. Yet, a new system may be touted as the most efficient for graph (or for log) data next week. How are they to tell the manager that *no, they are not going to migrate the application to that system*? What, if any, part of the data to deploy there? Would it be faster? Who knows?

In this work, we present ESTOCADA, a platform we have started building, to help *deploy* applications having to deal with *mixed-model data*, relying on a *dynamic set of diverse, heterogeneous data stores*. While heterogeneous data integration is an old topic [5,8,14,16], the remark "one-size does not fit all" [18] has been revisited [11,15], and the performance advantages brought by multi-stores have been recently noted e.g., in [13]. The set of features which, together, make ESTOCADA novel are:

Natively multi-model ESTOCADA supports a variety of data models, including flat and nested relations, trees and graphs, including important classes of semantic constraints such as primary and foreign keys, inclusion constraints, redundancy of information within and across distinct storage formats, etc. which are needed to enforce application semantics.

Application-invisible ESTOCADA provides to client applications access to each dataset in its native format. This does not preclude other mapping / translation logic above ESTOCADA' client API but we do not discuss them in this paper. Instead, our focus is on efficiently storing the data, even if in a very different format from its original one, as discussed below.

Fragment-based store ESTOCADA stores each dataset as a set of fragments, whose content may overlap. The fragmentation is completely transparent to ESTOCADA' clients, i.e., it is the system's task to answer queries based on the available fragments.

Mixed store Each fragment may be stored in any of the stores underlying a ESTOCADA installation, be it relational, tree- or graph-structured, based on key-value pairs etc., centralized or distributed, disk- or memory-based etc.

View-based rewriting and view selection The crucial ingredient to meeting our requirements is *view-based rewriting with constraints*. Specifically, each data fragment is internally described as a materialized view over one

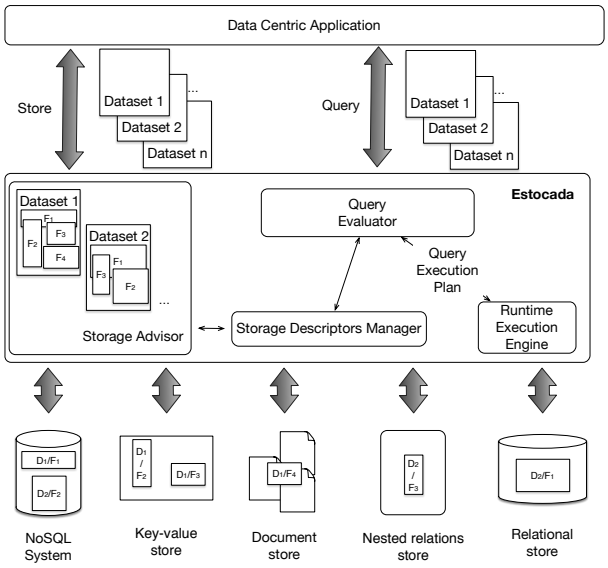


Fig. 1. ESTOCADA architecture.

or several datasets; query answering amounts to view-based query rewriting. Describing the stored fragments as views over the data allows changing the set of stores with no impact on ESTOCADA' applications [8]; this simplifies the migration nightmare outlined above.

In the sequel we present ESTOCADA's architecture, and walk the reader through the main technical elements of our solution, by means of an example. Finally, we discuss related works and conclude.

2 Architecture

The architecture we envision is depicted in Figure 1. Data centric applications issue two types of requests to ESTOCADA: *storing* data, and *querying* data.

The *Storage Advisor* (SA) module is in charge of splitting each dataset D_i into possibly overlapping fragments D_1/F_1, ..., $D_1/F_n, D_2/F_1, \ldots$, and delegating their storage to one of the underlying *data management systems*. In Figure 1, this is illustrated by a NoSQL store, a key-value store, a document store, one for nested relations, and finally a relational one. For each fragment D_i/F_j residing in the store S_k, the SA generates a *storage descriptor* $sd(S_k, D_i/F_j)$. The descriptor specifies *what* data (the fragment D_i/F_j) is stored *where* within S_k.

The *Storage Descriptor Manager* (SDM) keeps the catalog of the available storage fragments, that is the set of the current storage descriptors. The SDM also records some cost information C_{sd} for each storage descriptor sd, characterizing the processing costs involved in accessing the fragment D_i/F_j through the data access operation encoded in sd.

The *Query Evaluator* (QE) receives queries (or data access requests) issued by the application, in the original language of a dataset D_i. To handle them, the evaluator looks up the storage descriptors corresponding to fragments (views) of

D_i and rewrites the data access requests using the available materialized views (or fragments). The evaluator then selects one among these rewritings considered to have the best performance.

The *Runtime Execution Engine* (REE) translates a rewriting (essentially a logical plan joining the results of various data access operations on the store) into a physical plan which can be directly executed by dividing the work between (*i*) the stores $S_1, \ldots, S_k$ and (*ii*) ESTOCADA's own runtime engine, which supplies implementations of physical operators such as select, join, etc.

3 Under the hood

ESTOCADA aims at efficiently managing several datasets across a set of heterogeneous stores. In this section we provide some details of the main issues raised by our approach: (*i*) uniformly describing data fragments stored in the heterogeneous stores we consider, by means of *storage descriptors*; (*ii*) *view-based query rewriting* to identify the best storage data fragments to be used in order to answer a data access request.

The presentation of the contents uses a toy example inspired from an application that is based on Open Data published and shared in a digital city context. The purpose of the application is to predict the traffic flow and the consequent customer behavior taking into account information about events that influence people behavior such as city events, weather forecasts, etc. The data used in the project comes from city administrations, public services (e.g., weather and traffic data), companies, and individuals in the area, through Web-based and mobile applications; the sources are heterogeneous, comprising RDF, relational, JSON, and key-value data.

3.1 Dataset fragment representation

We assume a first dataset D of structured documents, such as XML, JSON etc., storing public transport information for the whole city area. The dataset is fragmented across three systems, as follows: a MongoDB document store holds tram and metro information; bus routes reside in a Redis key-value store, while RER and metro routes are stored within PostgreSQL. In the following, we discuss details of these fragments' storage, illustrating the expressive power of their storage descriptors.

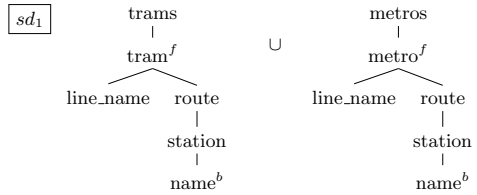
MongoDB fragment Fragment F_1 comprises trams and metros, stored in a MongoDB collection named `trams_metros`. Each line is modeled as a document. A document contains the line name and the route, that is, the names of all the stops; the latter are stored as an array reflecting their order. Thus, the *what* part of the storage descriptor sd_1 corresponding to F_1 is the query:

$$D/\texttt{trams/tram} \cup D/\texttt{metros/metro}$$

expressed in a simple XPath-like syntax that we shall use throughout this section. In practice, such view-defining queries are written in the native language of the dataset D. To access the data, MongoDB provides the FIND operation that, given a station name s, retrieves the documents describing all the lines whose route includes s:

```
db.trams_metros.find( { route: s } )
```

In the above, `route:` s encodes the constraint that the station s appears in the metro or tram route. Thus, the *how* part of sd_1 is depicted in the following figure, where the b superscript reflects that the value of that node must be provided (the node is said to be *bound*), while f denotes the node(s) whose content is obtained by the respective access method, and which are said *free* [17].



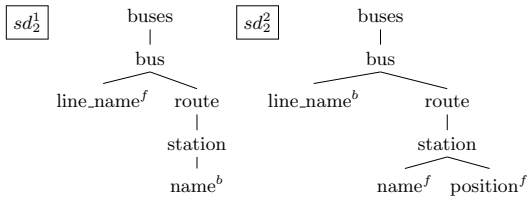
The above example illustrates a valuable feature of the *how* storage descriptor component, namely *binding patterns*. These are very useful especially when describing access methods supported by efficient current-day stores, such as key-value stores or document stores providing built-in search functionality, as MongoDB in the above example. Other lower-level information typically found in the *how* part includes the name of the database `db`, access credentials to the database, etc. The final component of sd_1 is a cost function C_1 which, given a station name s_1, estimates the cost of retrieving from MongoDB all the lines passing through station s_1.

Redis fragments comprise bus information. The data can be accessed in two different ways (the *how* part), leading to two storage descriptors: sd_2^1 and sd_2^2, as we explain next.

First, the bus information is broken down into pairs of the form $(s|n)$, where s denotes a station name and is used as a Redis key, whereas n is a bus line name stored as a value associated to s (Redis allows storing a set of values on a given key). Thus, the *what* part of sd_2^1 can be encoded by the query q_2^1, describing F_2^1 as a materialized view over D:

```
for $b in D/buses/bus
return ($b/name, $b/route/station)
```

As for the *how* part, F_2^1 data can be accessed by providing values for the station name, and receiving bus line names in return. This is represented by the left-hand tree with a binding pattern:



The same bus data is split again in a different way within Redis: using the bus line name n as a key, to which we associate a value for each station in the

route, by appending the name of each station to its order along the route. The corresponding query q_2^2 is:

```
for $b in D/buses/bus, $s in $b/route/station
return ($b/name as bname, $s/name as sname, $s/position())
```

where the `position` function is used to record in the (unordered) Redis store the position of each station along the route. As for the *how* part, in sd_2^2 the binding pattern is the one shown at right of the previous figure.

Cost descriptors C_2^1, C_2^2, and Redis access information (omitted here) complete the storage descriptors sd_2^1 and sd_2^2.

Postgres fragments Regional train and metro information is stored in Postgres, under the form of the following five tables (underlined attribute denote primary keys):

`Train(`$\underline{\texttt{rid, rname}}$`)` `Metro (`$\underline{\texttt{mid, mname}}$`)` Station($\underline{\text{sid}}$, sname)
`Tstat(`$\underline{\texttt{rid, sid}}$`, pos)` `Mstat (`$\underline{\texttt{mid, sid}}$`, pos)`

The query defining the first relation as a view over D is:

```
for $t in D/trains/train return ($t/id(), $t/name)
```

The four other queries are similar and we omit them for brevity. Regarding the *how* part of the storage, each table can be scanned, which is reflected by five storage descriptors $\boxed{sd_3^1}$ to $\boxed{sd_3^5}$, one for each table, e.g., `Train(tid`^{f}`, tname`^{f}`)`. Further, assuming that an index exists on `Station.sname`, this is modeled by the storage descriptor $\boxed{sd_3^6}$ with the binding pattern `Station(sid`^{f}`, sname`^{b}`)`. The presence of indexes is one reason why we create a distinct fragment (and storage descriptor) for each binding pattern: the cost is likely to be very different for an indexed access through sd_3^6, than by the respective scan-based descriptor sd_3^5. The second reason is that each fragment is usable only when the values of the attributes bound in its binding pattern can be filled in (from the query or another already accessed fragment).

3.2 View-based query rewriting

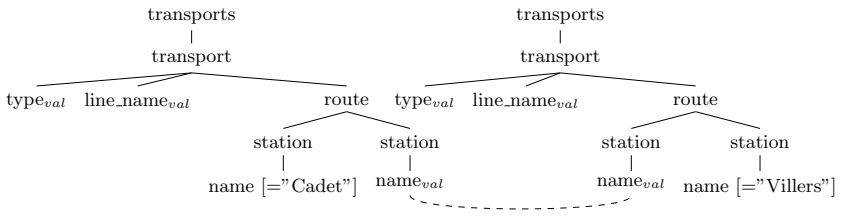


Fig. 2. Sample query.

Following with our example, we consider an application query, shown in Figure 2, asking for all transportation paths going from station "Cadet" to station "Villers" with at most one connection. In Figure 2, *val* subscripts denote nodes

to be returned by the query, while the dashed line denotes a join. Part of the answers to this query can be obtained by joining sd_1 (accessed using the value "Cadet" for the bound station name) with itself (accessed a second time using the value "Villers"). The intersection of the results (routes) allows finding common (connection) stations, thus providing paths made of two trams, two metros, or one of each. To obtain tram-bus or metro-bus paths, one may join sd_1 (accessed as above with "Cadet") with sd_2^1 for each station in the routes retrieved from sd_1, to learn the lines on which the connecting station sits, and then with sd_2^1 again (accessed with "Villers"), to retain those bus lines that stop at Villers. An alternative way to obtain such paths consists in accessing sd_2^1 to learn the bus lines that stop at Villers, finding their routes by accessing sd_2^2, and then further intersecting these with routes that pass through Cadet, as retrieved from sd_1. Further, joining sd_1 (accessed with "Cadet") with sd_3^1, sd_3^3, and sd_3^5 and then selecting on "Villers" provides tram-train and metro-train paths. The same paths can be obtained by rather employing the more efficient sd_3^6. Metro-train or metro-metro paths can also be obtained by solely relying on the corresponding Postgres fragments, etc. Combining such partial answers through unions leads to a large number of (equivalent) rewritings of the query; the one with the least estimated cost is selected for execution.

Thus, query answering in ESTOCADA reduces to a problem of cost-based query reformulation under constraints. First, all view definitions (*what* part of the storage descriptors) are compiled into an internal, conjunctive query model with constraints. The constraints are (*i*) application-driven, i.e., those holding on the original dataset, or (*ii*) inserted by the translation to correctly account for the structural characteristics of the original data model.

Cost-based reformulation under constraints of this expressivity and scale has not been addressed by any commercial system; the only relevant research prototype we are aware of is [10], which we adopt as a starting point.

4 Related Work and Conclusions

The interest of simultaneously using multiple data stores has previously been noted in [11,15]. The system we propose follows this direction and is also more general than the recently proposed [13] that demonstrated the performance benefits of using only two different systems. Our work also shares some features of classic [8,14] and recent [2] data integration or mediator systems, by dividing query processing between underlying stores and a runtime integration component running on top of them.

Adaptive stores have been the focus of many works such as [1,4,9,12]. The novelty of ESTOCADA here is to support multiple data models, by relying on powerful query reformulation techniques under constraints.

View-based rewriting and view selection techniques are grounded in the seminal works [8,14]; the latter focuses on maximally contained rewritings, while we target exact query rewriting, which leads to very different algorithms.

ESTOCADA aims at introducing materialized views and indexes both to integrate and to get the best performance out of a heterogeneous data store starting

from complete and scalable algorithms for query reformulation under constraints, such as [10] and evolving them appropriately.

Finally, our approach shares some analogies with works in data exchange such as Clio [6,7] but ESTOCADA aims at providing to the applications transparent data access to heterogeneous systems, relying on fundamentally different rewriting techniques.

To conclude, we believe that hybrid (multi-store) systems can bring very significant boost to performance; further, such systems must accommodate dynamic sets of stores, and adapt to changing workloads. These two aspects lead to local-as-view integration and view-based rewriting as cornerstones of ESTOCADA's approach; reformulation under constraints is required to guarantee correctly computed answers from a variety of stores. The implementation of ESTOCADA is ongoing, building on [10] and our experience in [12].

Acknowledgments This work has been partially funded by the Datalyse "Investissement d'Avenir" project, by the associated INRIA-Sillicon Valley OAK-SAD team, and the KIC ICT Labs Europa activity.

References

1. I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: a hands-free adaptive store. In *SIGMOD*, 2014.
2. P. Atzeni, F. Bugiotti, and L. Rossi. Uniform access to NoSQL systems. *Information Systems*, 2014.
3. F. Bugiotti, D. Bursztyn, A. Deutsch, I. Ileana, and I. Manolescu. Invisible Glue: Scalable Self-Tuning Multi-Stores. In *CIDR*, 2015.
4. D. Dash, N. Polyzotis, and A. Ailamaki. Cophy: A scalable, portable, and interactive index advisor for large workloads. *PVLDB*, 4(6):362–372, 2011.
5. A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *VLDB*, 2003.
6. R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, 2003.
7. L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD*, 2005.
8. A. Y. Halevy. Answering Queries Using Views: A Survey. *VLDB Journal*, 2001.
9. S. Idreos, M. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.
10. I. Ileana, B. Cautis, A. Deutsch, and Y. Katsis. Complete yet practical search for minimal query reformulations under constraints. In *SIGMOD*, 2014.
11. A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. WWHow! Freeing Data Storage from Cages. In *CIDR*, 2013.
12. A. Katsifodimos, I. Manolescu, and V. Vassalos. Materialized view selection for XQuery workloads. In *SIGMOD*, 2012.
13. J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, J. Tatemura, N. Polyzotis, and M. Carey. MISO: souping up big data query processing with a multistore system. In *SIGMOD*, 2014.
14. A. Levy, A. Rajaraman, and J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *VLDB*, 1996.
15. H. Lim, Y. Han, and S. Babu. How to Fit when No One Size Fits. In *CIDR*, 2013.
16. I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries on heterogeneous data sources. In *VLDB*, 2001.
17. A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.
18. M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*, 2005.