Invisible Glue: Scalable Self-Tuning Multi-Stores

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

Invisible Glue: Scalable Self-Tuning Multi-Stores

Francesca Bugiotti
INRIA & U. Paris-Sud, France
francesca.bugiotti@inria.fr

Damian Bursztyn
INRIA & U. Paris-Sud, France
damian.bursztyn@inria.fr

Alin Deutsch
UC San Diego, USA
alin@cs.ucsd.edu

Ioana Ileana
Telecom ParisTech, France
ioana.ileana@telecom-paristech.fr

Ioana Manolescu
INRIA & U. Paris-Sud, France
ioana.manolescu@inria.fr

ABSTRACT

Next-generation data centric applications often involve diverse datasets, some very large while others may be of moderate size, some highly structured (e.g., relations) while others may have more complex structure (e.g., graphs) or little structure (e.g., text or log data). Facing them is a variety of storage systems, each of which can host some of the datasets (possibly after some data migration), but none of which is likely to be *best for all, at all times*. Deploying and efficiently running data-centric applications in such a complex setting is very challenging.

We propose ESTOCADA, an architecture for efficiently handling highly heterogeneous datasets based on a dynamic set of potentially very different data stores. ESTOCADA provides to the application/programming layer access to each data set *in its native format*, while hosting them internally in a set of *potentially overlapping fragments, possibly distributing (fragments of) each dataset across heterogeneous stores*. Given workload information, ESTOCADA self-tunes for performance, i.e., it automatically choses the fragments of each data set to be deployed in each store so as to optimize performance. At the core of ESTOCADA lie powerful view-based rewriting and view selection algorithms, required in order to correctly handle the features (nesting, keys, constraints etc.) of the diverse data models involved, and thus to marry correctness with high performance.

1. CONTEXT AND OUTLINE

Digital data is being produced at a fast pace and 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 just fine on the relational data. However, the social graph data fits badly in that system, and the company attempts to store it 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 [14] 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!

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. Part of the log data has been moved in the in-memory column store, too, when the social data was stored there; this made their joint exploitation faster. But the whole log dataset could not fit in the single-node column store installation, and data migration fatigue had settled in before a suggestion was made (and rejected) to move everything to yet another cluster installation of the column store. The team working on the application feels battered and confused. The application is sometimes very slow. Migrating data is painful at every change of system; they are not sure the complete data set 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

7th Biennial Conference on Innovative Data Systems Research (CIDR '15) January 4-7, 2015, Asilomar, California, USA.

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

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 and self-tune for performance* 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 [16, 9, 5, 19], the remark "one-size does not fit all" [22] has been revisited for instance in the last CIDR [18, 12, 6], and the performance advantages brought by multi-stores have been recently noted e.g., in [14]. Self-tuning stores have also been a topic of hot research. 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 Each data set is stored 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. Query answering must be aware of the *constraints* introduced implicitly when storing fragments in non-native models. For instance, when tree-structured data are stored in a relational store, the resulting edge relation satisfies the constraint that each node has at most one parent, the descendant and ancestor relations are inverses of each other and are related non-trivially to the edge relation, etc.

Self-tuning store The choice of fragments and their placement is fully automatic. It is based on a combination of heuristics and cost-based decisions, taking into account data access patterns (through queries or simpler data access requests) as these become available.

View-based rewriting and view selection The invisible glue holding all the pieces together is *view-based rewriting with constraints*. Specifically, each data fragment is internally described as a materialized view over one or several datasets; query answering amounts to view-based query rewriting, and storage tuning relies on view selection. Describing the stored fragments as views over the data allows changing the set of stores with no impact on ESTOCADA' applications [9]; this simplifies the migration nightmare outlined above. Finally, our reliance on views gives sound foundation to efficiency, as it guarantees the complete storage of data, and the correctness of the fragmentation and query answering, among others.

Technical challenges The ESTOCADA scenario involves the coexistence of a large number of materialized views mixing data formats (modeling the native sources) with significant redundancy between them (due to repeated migration and view selection arising organically over the history of the system, as opposed to clean-slate planning). While the problem of rewriting using views is classical, it has typically been addressed and practically implemented only in limited scenarios that do not apply here. These scenarios feature (*i*) only relatively small numbers of views; (*ii*) minimal overlap between views as their selection is planned ahead of time; (*iii*) views expressed over the same data model; (*iv*) rewriting that exploits only limited integrity constraints (typically only key/foreign key in existing systems). The large number of views and their redundancy notoriously contribute (at least) exponentially to the explosion in the search space for rewritings, even when working within a single data model.

In the sequel, we introduce some motivating scenarios, 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, then conclude.

2. APPLICATION SCENARIOS

We now present two typical scenarios which stand to benefit from our proposal. They are inspired from real-world applications being built by several partners including us, within the French R&D project Datalyse on Big Data analytics. (`http://www.datalyse.fr`).

Open Data warehousing for Digital Cities. The application is based on Open Data published and shared in a digital city context. The purpose 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 (e.g., celebrations, demonstrations, or a highly attended show or football match), weather forecasts (bad weather often leading to traffic slowdown), etc. The analysis is performed in a metropolitan area of a 600.000-strong French city. 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. More precisely, (*i*) Open data about traffic and events is encoded in RDF; (*ii*) city events information is published as JSON documents; (*iii*) social media data holding users' locations, as well as their notifications of public transport events (such as delayed buses or regional trains) is organized in key-value pairs; (*iv*) weather data is generated as relational tuples.

The application comprises the following queries:

- Estimated concentration of people and vehicles in a particular area and moment. One use of such information is to plan some corrective/preventive measures, for instance diverting traffic from an overcrowded area; it can also be used to identify business opportunities

according to where people are. This query requires: traffic information, social events, social network information, the weather (open space events may be cancelled depending on the weather) and public transport information.

- Most popular trajectories in the area which can be used to optimize public transport or car traffic routes. This needs traffic information, social network data on traffic events, and public transportation timetables.

Large-scale e-commerce application. We consider a scenario from a large-scale e-commerce application whose goal is to maximize sales while improving the customer experience. A large retailer wants to use the clients' social network activity with the e-commerce application logs, to improve targeted product recommendation. This requires exploiting the data produced by the users both actively (orders, product reviews, etc.) and passively (logs), as well as social network data.

The heterogeneous sources of information in this scenario are: (*i*) shopping cart data stored in key-value pairs; (*ii*) the product catalog, structured in documents using a data store with full-text search, faceted search and filtering capabilities; (*iii*) orders, stored in a relational data store; (*iv*) user data (such as birth date, gender, interests, delivery addresses, preferences, etc.) organized in documents; (*v*) products reviews and ratings, structured in a partitioned row store, and (*vi*) social network data organized in key-value pairs. The queries in this scenario:

- Retrieve items to recommend to each user, by displaying them on the user homepage and inside bars while the user is shopping. This requires combining cart information, the product catalog, past relational sale data, personal user data, product reviews, and possible recommendations gleaned from the social network.

- Improve product search: this requires attribute-based search in the product catalog, as well as user's recent search and purchase history to decide which products to return at the top of the search result.

This application uses massive data (the sale history of many users), yet response time is critical, because query answers must be made available in the user's Web interface. To get such performance, the engineers in charge of the application have decided to use memcached [23] to make access to *parts* of the database very fast. However, automatically deciding which parts to put in memcached, and correctly computing results out of memcached and the other external sources is challenging, especially given the heterogeneity of the data representation formats.

3. ARCHITECTURE

The architecture we envision is depicted in Figure 1. Data centric applications issue two types of requests to ESTOCADA: *storing* data, and *querying*, or, more generally, issuing data access requests, through each data set's native query language or API[2] As previously mentioned, we mostly target applications based on many data sets $D_1, D_2, \ldots, D_n$,

[2]As previously said, one could integrate *on top* of ESTOCADA the various data sources under a common data model; this is out of the scope of the architecture we describe here.

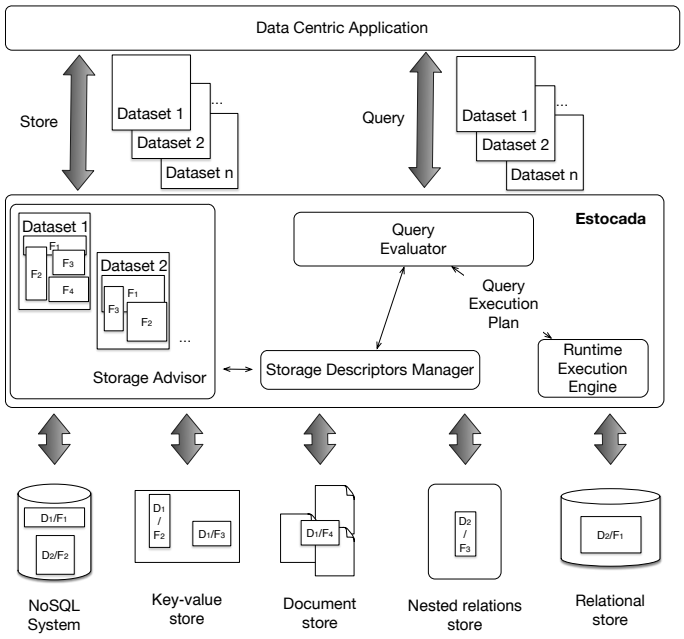


Figure 1: Estocada architecture.

even though our smart storage method may be helpful even for a single data set, distributing it for efficient access across many stores, potentially based on different data models.

The *Storage Advisor* module is in charge of splitting each data set into possibly overlapping fragments D_1/F_1, D_1/F_2, ..., $D_1/F_n, D_2/F_1, \ldots$, etc., and delegating their storage to one of the underlying *data management systems*. In the figure, this is illustrated by a NoSQL store, a key-value store, a document store, one for nested relations, and finally a relational one. Several among the available stores could be based on the same data model; each of them may be distributed or centralized, memory- or disk-resident, etc.

The data set fragmentation decisions are taken based on (*i*) information about the workload of data access requests, to be issued by the applications. This may comprise *exact* data access requests, or *parameterized* ones, where the values of some parameters (typically constants used in a selection) are not statically known but represented by a parameter; (*ii*) a set of heuristics driven by the data models of the dataset D_i, and of the store S_k holding D_i/F_j.

For each data fragment D_i/F_j residing in the store S_k, a *storage descriptor* $sd(S_k, D_i/F_j)$ is produced. The descriptor specifies *what* data (the fragment D_i/F_j) is stored *where* within S_k. The *what* part of the descriptor is specified by a query over the data set D_i, following the *native model* of D_i. The fragment can thus be seen as a *materialized view* over D_i. The *where* part of the descriptor is structured according to the organization of data within S_k. For instance, if S_k is a relational store, the *where* information consists of the schema and table name, whereas if S_k is a key-value store, it could hold the name of the collection, the attribute name, etc.Finally, the descriptor $sd(S_k, D_i/F_j)$ also specifies the data access operation supported by S_k which allows retrieving the D_i/F_j data (such as: a table scan, a look-up based on a collection name, column group name, and column name

in a key-value store, etc.), as well as the access credentials required in order to connect to the system and access it.

The *Storage Descriptor Manager* (SDM, in short) 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 cost information can be gathered explicitly using the tools provided by the storage system S_k, and/or inferred by ESTOCADA dynamically over time by monitoring execution and refining the values of its own cost model. For instance, the cost associated to a descriptor sd referring to a full collection (or table) scan will be different from the one describing an index- (or key-based) lookup access.

The SDM also records the usage frequency (and cost) of each storage descriptor. Based on this information, the *Storage Advisor* may recommend dropping a fragment rarely used over a period of time and/or slow-performing, or ask for adding a new fragment which fits recent, frequent data access requests. Observe that this reorganization of storage, under the control of the Storage Advisor, spans over all the data stores. Thus, a particular case of data reorganization is moving a fragment D_i/F_j from one store to another.

The *Query Evaluator* receives queries (or data access requests) issued by the application, in the original language (model) of a data source D_i. To handle them, the evaluator looks up the storage descriptors corresponding to fragments of D_i, and computes ways to answer the incoming request based on the available fragments; thus, the evaluator rewrites the data access requests using the materialized views (or fragments). The evaluator then selects one among these rewritings considered to have the best performance, let us denote it $r(sd_1, sd_2, \ldots, sd_k)$, where each sd_i is a storage descriptor representing a fragment within the storage system S_i, and hands it to the runtime component (see below).

The *Runtime Execution Engine* 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. While in classical mediator style, the runtime of ESTOCADA can be used to combine data across different stores, we stress that this may be needed even for two fragments within the same store, if the store does not support joins internally; this is the case, for instance, of current key-value stores not providing native joins. Evaluating the physical plan returns results to the client application.

From the above description, it is easy to see that ESTOCADA resembles wrapper-mediator systems, where data resides in various stores and query execution is divided between the mediator and the wrappers. Different from mediators, however, ESTOCADA *distributes the data across the different-data model stores*, which are not autonomous but treated as slave systems, in order to obtain the best possible performance from the available combination of systems.

4. UNDER THE HOOD

ESTOCADA aims at efficiently managing several datasets across a set of heterogeneous stores. In this section, based on a toy example inspired from our Digital City Open Data Warehousing scenario, 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; (*iii*) *storage tuning* that moves data across systems and/or builds new data access structures for performance.

4.1 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.

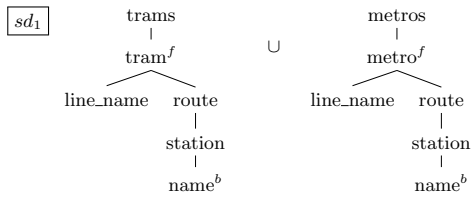
MongoDB fragment Fragment F_1 comprises trams and metros, stored in a MongoDB collection named `trams_metros`. Each line is modeled as a document. A document contains the line name and the route, that is, the names of all the stops; the latter are stored as an array reflecting their order. Thus, the "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* [20].



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. This may

be estimated once for all stations, or be stored at a finer granularity (different values for distinct s_1 bindings), etc.

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, in Figure 2.

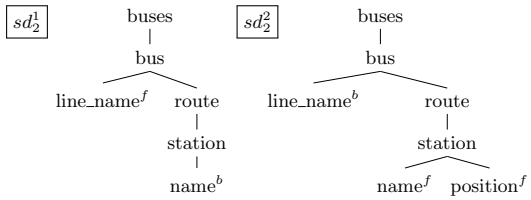


Figure 2: Sample storage descriptor.

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 in Figure 2.

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(`<u>`rid`</u>`, rname)` `Metro (`<u>`mid`</u>`, mname)`
`Tstat(`<u>`rid, sid`</u>`, pos)` `Mstat (`<u>`mid, sid`</u>`, pos)`

`Station(`<u>`sid`</u>`, sname)`

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).

DFS fragments A second dataset D' of social media notifications of *public transport events* is in a distributed file system (DFS). Events are partitioned in the DFS according to their date. Each event notification comprises the date, time, station, line, and a short description (e.g., delay on the line etc.). The data can be freely accessed through a (parallel) scan. We omit the query defining this fragment F_4 and its (all-f) binding pattern. Further, *city events* (such as concerts, public manifestations etc.) are also stored in the DFS, in two ways: partitioned by *event type* (fragment F_5^1) and by event date (fragment F_5^2). Fragments F_4, F_5^1, and F_5^2 illustrate a final feature of the "how" part of storage descriptors: *if* the data is partitioned, the SD records the partitioning attribute(s), in our case, date for F_4 and F_5^2 and event type for F_5^1. This information is extremely helpful during cost-based query rewriting (discussed below), as in large-scale distributed stores (e.g., based on MapReduce), joins which can be performed on common partitioning keys (for instance, join transport and city events *on the date* to find which city events lead to most transport disruption) are the most efficient [17].

4.2 View-based query rewriting

Following with our example, we consider an application query, shown in Figure 3, asking for all transportation paths going from station Cadet to station Villers with at most one connection. In Figure 3, *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 descrip-*

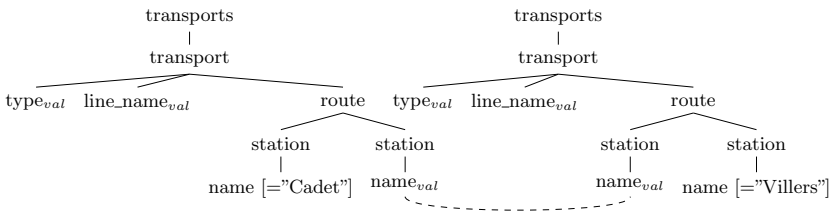


Figure 3: Sample query.

tors) 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. For instance, if data was originally structured in documents, the relational queries defining the views must be interpreted together with constraints stating e.g., that each document node has only one parent, any child node is also a descendant, etc. 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 [11], which we adopt as a starting point.

4.3 Unified Storage Tuning

Ultimately, ESTOCADA aims at making applications efficient by providing storage structures best adapted to the workload, based on any of the available stores. In our example, the storage advisor may recommend to move the metro data into Redis due the lower access costs and the application high demand for metro routes. If sd_1 is accessed multiple times using the station name, it can recommend an index on station names within MongoDB. If our sample query is run many times with different starting and ending stations, the storage advisor can recommend to materialize in a key-value store a view (asking for all the possible paths from one station to another with at most one change), with the name of the start station, for instance, as key.

The problem of automatic storage tuning, or cost-based, workload-driven index/materialized view recommendation, is long-known to be difficult due to the huge space of alternatives. So far, the problem has been studied for simpler situations when the views and the stores share the same data model. Heuristics such as avoiding views computable from others, or pruning candidate views based on the estimated costs of rewriting have been shown useful [13], as well as linear programming techniques to efficiently search through the alternatives [4]. In our setting, the choice of a store for a candidate view (fragment) also complicates the search: while there are more alternatives, all do not make sense, as some views (identified by queries in our internal language) may not be valid structures to build within some stores.

5. RELATED WORK

The interest of simultaneously using multiple data stores has previously been noted in [18, 6, 12]. In this paper, we propose a storage approach that allows *and optimizes* the simultaneous use of multiple data stores. Recent work also demonstrated the performance benefits of using two heterogeneous systems, in particular by materializing workload-driven views into the most efficient among the two [14, 15]. Our approach is more general, since ESTOCADA can take advantage of any number of storage systems, based on a variety of data models; this is supported by its central algorithm for query reformulation under constraints.

Our work shares some features of classic data integration or mediator systems, by dividing query processing between underlying stores and a runtime integration component running on top of them; recently, the integration of so-called "NoSQL" stores has been revisited e.g., in [3]. However, the automatic cross-model, cross-system storage tuning problem we consider has not been studied in mediator systems.

Adaptive stores have been the focus of many works such as [10, 2, 4, 13]. 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 [9, 16]; 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 methodologies are similar to the ones described in [5, 19] which did not consider storage tuning. Views are used in [21, 1] to improve the performance of a distributed relational system. In contrast, ESTOCADA aims at introducing materialized views and indexes to get the best performance out of a heterogeneous data store, leading first, to integrity constraints for modeling such heterogeneity, and second, a potentially very large number of views. To achieve correctness and practically relevant performance in such setting, we start from complete and scalable algorithms for query reformulation under constraints, such as [11], evolving them appropriately.

Finally, our approach shares some analogies with works in data exchange such as Clio [7, 8], allowing to migrate data between two schemas and to build completely defined mappings given a set of user-defined correspondences. In contrast, ESTOCADA aims at providing to the applications transparent data access to heterogeneous systems, relying on fundamentally different rewriting techniques.

6. CONCLUSIONS

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 [11] and our experience in [13].

Acknowledgments This work has been partially funded by the Datalyse "Investissement d'Avenir" project, by the as-

sociated INRIA-Sillicon Valley OakSad team, and the KIC ICT Labs Europa activity.

7. REFERENCES

[1] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous View Maintenance for VLSD Databases. In *SIGMOD*, 2009.

[2] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: a hands-free adaptive store. In *SIGMOD*, 2014.

[3] P. Atzeni, F. Bugiotti, and L. Rossi. Uniform access to NoSQL systems. *Information Systems*, 2014.

[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] J. Dittrich. Say No! No! and No! In *CIDR*, 2013.

[7] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, 2003.

[8] 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.

[9] A. Y. Halevy. Answering Queries Using Views: A Survey. *VLDB Journal*, 2001.

[10] S. Idreos, M. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.

[11] I. Ileana, B. Cautis, A. Deutsch, and Y. Katsis. Complete yet practical search for minimal query reformulations under constraints. In *SIGMOD*, 2014.

[12] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. WWHow! Freeing Data Storage from Cages. In *CIDR*, 2013.

[13] A. Katsifodimos, I. Manolescu, and V. Vassalos. Materialized view selection for XQuery workloads. In *SIGMOD*, 2012.

[14] 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.

[15] J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, J. Tatemura, N. Polyzotis, and M. J. Carey. Opportunistic physical design for big data analytics. In *SIGMOD*, 2014.

[16] A. Levy, A. Rajaraman, and J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *VLDB*, 1996.

[17] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using MapReduce. *ACM Comput. Surv.*, 46(3):31, 2014.

[18] H. Lim, Y. Han, and S. Babu. How to Fit when No One Size Fits. In *CIDR*, 2013.

[19] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries on heterogeneous data sources. In *VLDB*, 2001.

[20] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.

[21] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A Distributed SQL Database That Scales. In *PVLDB*, 2013.

[22] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*, 2005.

[23] Memcached distributed memory object caching system. http://memcached.org, 2012.