# Estocada: Stockage Hybride et Ré-écriture sous Contraintes d'Intégrité

Rana B. Al-Otaibi, Francesca Bugiotti, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, Stamatis Zampetakis

# Estocada: Stockage Hybride et Ré-écriture sous Contraintes d'Intégrité

Rana B. AL-Otaibi
UC San Diego,
USA
ralotaib@eng.ucsd.edu

Francesca Bugiotti
CentraleSupélec & INRIA,
France
francesca.bugiotti@inria.fr

Damian Bursztyn
INRIA & École Polytechnique,
France
damian.bursztyn@inria.fr

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

Ioana Manolescu
INRIA & École Polytechnique,
France
ioana.manolescu@inria.fr

Stamatis Zampetakis
INRIA,
France
stamatis.zampetakis@inria.fr

## ABSTRACT

La production croissante de données numériques a conduit à l'émergence d'une grande variété de systèmes de gestion de données (Data Management Systems, ou DMS). Dans ce contexte, les applications à usage intensif de données ont besoin *(i)* d'accéder à des données hétérogènes de grande taille ("Big Data"), ayant une structure potentiellement complexe, et *(ii)* de manipuler des données de façon efficace afin de garantir une bonne performance de l'application. Comme ces différents systèmes sont spécialisés sur certaines opérations mais sont moins performants sur d'autres, il peut s'avérer essentiel pour une application d'utiliser plusieurs DMS en même temps.

Dans ce contexte nous présentons ESTOCADA, une application donnant la possibilité de tirer profit simultanément de plusieurs DMSs et permettant une manipulation efficace et automatique de données de grande taille et hétérogènes, offrant ainsi un meilleur support aux applications à usage intensif de données. Dans ESTOCADA, les données sont reparties dans plusieurs fragments qui sont stockés dans différents DMSs. Pour répondre à une requête à partir de ces fragments, ESTOCADA est basé sur la re-écriture de requêtes sous contraintes; ces dernières sont utilisées pour représenter les différents modèles de données et la répartition des fragments entre les diffèrents DMSs.

## 1. INTRODUCTION

There is significant consensus around the observation that the times where one system fits all data management needs are over [24]. Nowadays data-intensive applications often involve dealing with diverse datasets in terms of size and structure: relations flat or nested, complex-structure graphs, documents, and poorly structured logs, or even text data. Processing tasks to be run this data are also very varied: selective or bulk processing, structure traversal and aggregation, joins, grouping, pattern matching, advanced analytic processing using dedicated functions, etc.

Facing these needs, a wide variety of DMSs is now available to be used in data management applications. These systems include structured data*base* management systems from major vendors, which currently come in centralized or cloud edition, supporting traditional relational stores (disk- or memory-resident), but also novel formats such as JSON, RDF, graphs, text, etc. They have been joined by the large crowd of so-called NoSQL systems, a very broad term encompassing at one end, novel architectures for the very fast processing of extremely simple, small-granularity data encoded in key-value pairs, and at the other end, large-scale platforms aiming at massive parallel computation, such as those adopting the Bulk Synchronous Parallel approach. Among these, the well-known MapReduce model has been extended with many more operators e.g., in Spark or Flink; many of its implementations lift the performance disadvantages of early Hadoop versions. More generally, numerous systems are competing for the glut of so-called "Big Data" applications; their capabilities (supported data format and operations) and their performance blueprint (in terms of speed and scale) makes each of them unique, and enable numerous optimization opportunities.

Further, observe that a given choice of storage systems may need to be changed over time, as the data or application needs change, as new more efficient system may become available, or on the contrary their usage needs to be discontinued (for instance due to changes in the application owner's IT policy, or in the pricing of a certain commercial system). In such cases, one should not have to modify (rewrite) the applications, but rather have it run and adapt seamlessly to the new context.

We propose to demonstrate ESTOCADA [5, 6], a platform providing applications with *transparent, optimized* data access to *diverse, heterogeneous storage systems.* ESTOCADA enables storing one dataset in a set of possibly overlapping fragments, while providing to the application access to this dataset in the native language most suited for the dataset, e.g., SQL if the data is relational (or object-relational), JSONiq if the data is in JSON documents, etc. At the heart of ESTOCADA lies a common modeling of the different data set and storage systems data models, data fragments, and also queries in an internal, expressive formalism based on re-


*BDA* 2016, 15 au 18 Novembre, Poitiers, France.

lational queries and constraints (which, as we show, is rich enough to capture rich data structure including nesting, object identity, functional dependencies and more); query processing then starts by solving a problem of query rewriting under constraints, backed by an efficient recent algorithm [15]. Demo attendees will have the opportunity to try ESTOCADA on a set of systems of very varied nature, data models, and architectures; they will use different data fragmentation and queries and inspect the resulting query evaluation plans.

## 2. MOTIVATING SCENARIO

We describe below a **large-scale online marketplace application scenario** which stands to benefit from our approach. It is inspired from a real-world application from the French R&D collaborative project Datalyse on Big Data analytics (`http://www.datalyse.fr`). The marketplace aims to maximize sales while improving the customer experience, by exploiting the data produced by the users both actively (orders, product reviews, etc.) and passively (browsing recorded in Web logs).

With respect to the *data model*, the product catalog is organized in JSON documents; user data (coordinates, payment information, etc.), order and shipping information are in a set of relations, shopping carts are documents, while data recording the users' interaction with the marketplace is in HTTP log files. After manually deploying and experimenting with a few different settings, the system's first release makes the following choices: product catalogs are stored in SOLR (providing full-text indexing and search based on Lucene), user accounts, preferences, orders and shipping are stored in a Postgres cluster, the MongoDB system is used to store the shopping carts while the logs are stored in a cluster and Spark is used to process it in parallel, retrieving information and statistics about users' visits on the Web site, etc.

After deploying and exploiting this architecture for a while, the development team noticed that *predominant queries* (for user preferences on one hand, and their shopping carts on the other) *correspond to key-based searches.* They decided to investigate the usage of the Voldemort key-value store, for storing the corresponding data fragments. This required: migrating these fragments into Voldemort (an error-prone process as data needs to be restructured in a different data model), adapting the application to interact with the key-value store instead of the document and relational stores previously employed; measuring the resulting performance and deciding which store to use (say, MongoDB). This change brought a performance gain of 20% on the application workload.

Subsequently, the personalized item search query became the bottleneck and required extra care; this query *combines user past purchases (from the relational store) and the browsing history log*, to identify the products which should be shown first in response to a user search. To speed up this query, it was suggested to materialize the result of the join between past purchases and browsing history data into a nested relation stored in Spark; further, this relation should be indexed by the user ID and product category. This change improved performance by an extra 40%, which was very well received. However, the satisfaction was short-lived, as business needs brought new query requirements, and serving these queries better appeared to conflict with the choices previously made. The team faces the option of re-migrating data and re-re-factoring the application; they decide not to do it (and thus miss the extra performance improvement) due to lack of manpower.

## 3. APPROACH AND ARCHITECTURE

We now explain how ESTOCADA is capable of automating the solution to scenarios such as the one previously described. The main focus of this work is on its ability to answer queries over an application dataset based on fragments stored in a variety of underlying DMSs, across distinct data models and platforms. The remaining problem is to automatically recommend the way in which the data sets that an application needs to work with should be fragmented across such DMSs in order to maximize performance under specific storage or cost parameters. In the present demo, we will present simple heuristics for the latter, since work is in progress, and focus on showcasing the expressive power and performance benefits that ESTOCADA can bring to query evaluation.

To adapt to changes in the datasets, workload, and set of DMSs being used, we chose to internally *represent each data fragment as a materialized view over one or several datasets*; thus, query answering amounts to view-based query rewriting. As is well-known from prior work in data integration, this local-as-view approach allows the application to remain unchanged as the underlying data collections are modified. Further, 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.

**Pivot model with constraints** To further simplify the development of applications, each dataset is accessed through a language specific to its native data model, be it SQL for relational stores, key-based search API for key-value data, etc. However, for efficiency, a fragment $F$ of a dataset $D$ (whose data model is $\mathcal{M}_D$) may be stored in a data model $\mathcal{M}_F$ different from $\mathcal{M}_D$; similarly, a fragment $F'$ may store combined results from different datasets of possibly different data models, leading to more cross-model transformation of the data between the application dataset and the stored fragments. To enable query rewriting over and across different data models, we translate into an *internal pivot model* the declarative specification of the data stored in each fragment, as well as the incoming query, formulated in the application dataset model; specifically, our pivot model is based on relational conjunctive queries. Further, to correctly account for the characteristics of each application data model $\mathcal{M}_a$ and storage data model $\mathcal{M}_s$, we describe their specific features in the same pivot model, by means of powerful *constraints.* For instance, we describe the organization of a document data model (whether this concerns $\mathcal{M}_a$ or $\mathcal{M}_s$) 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.[1]

More generally, constraints allow a faithful internal modeling of datasets, since they can express functional dependencies and keys (for instance, node or tuple IDs) naturally present in many settings, be it relations, documents or graph

[1]Such modeling had first been introduced in local-as-view data integration XML integration works [8,22]; see also Section 5.

stores. Also, importantly for the usage of key-value stores, we rely on an original *encoding of access pattern restrictions* such as "the value of the key must be specified in order to access the values associated to this key" into relations with constraints. This enables building only *feasible* rewritings, i.e., such that the information needed to access a given data source is either provided by the query, or has been obtained from data sources previously accessed while evaluating the rewriting.

To rewrite queries in the presence of constraints, the method of choice is known as Chase & Backchase (C&B, in short), a classical powerful tool long considered too inefficient to be of practical relevance. ESTOCADA exploits the very significant performance savings brought by the recent provenance-aware C&B algorithm (PACB, in short) [15]. PACB drastically reduces the back-chase effort by keeping track of the results of the various chase steps applied during the algorithms, to avoid repeated and fruitless work; this results in rewriting speedups that can even outperform a commercial relational optimizer by 1-2 orders of magnitude (in terms of combined optimization and execution time).

**Making rewritings executable** From the above, it follows that query rewriting takes place, first, at the level of our pivot relational conjunctive model endowed with constraints, and it leads to a rewriting which is a conjunctive query over the relations corresponding to the stored fragments.

Depending on the data model of these fragments, the relational atoms used in the rewritings may either correspond to actual relations, or to key-value collections which can be seen as relations with binding patterns, or to the virtual relations used to encode more complex data models, such as the Node, Child and Descendant relations mentioned above (the encoding of nested relations such as supported e.g., in Pig and HBase is very similar). From this relational, conjunctive rewriting, a *rewriting translation step* is performed to: (*i*) group the rewriting atoms referring to each distinct fragment involved in the rewriting; for instance, it can be inferred that the three atoms *Document(dID, "file.json"), Root(dID, rID), Child(rID, cID), Node(cID, book)* found in a rewriting refer to a single document, by following the connections among nodes and knowledge of the JSON data model; (*ii*) reformulate each such rewriting snippet into a query which can be completely evaluated over a single fragment; (*ii*) if several fragments are stored in the same underlying DMS, identify the largest subquery that can be delegated to that DMS, along the lines of query evaluation in wrapper-mediator systems. Observe that if the DMS has a distributed architecture, e.g., Spark deployed on a cluster, the delegated subquery will be evaluated in parallel fashion, allowing ESTOCADA to leverage its efficiency.

**Evaluation of non-delegated operations** Rewriting translation may be unable to push (delegate) some query operations to the DMS storing a fragment if the DMS 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 (such as JSON or XML documents, or nested tuples), and if the inputs to this operation are not stored in a DMS supporting such result construction natively, it will have to be executed outside of the underlying DMSs. To evaluate such "last-step" operations, ESTOCADA comprises its own *lightweight execution engine*, based on a nested relational model, whose atomic types include constants, node IDs, and document types; it provides in particular implementations of the BindJoin operator needed to access data sources with access restrictions.

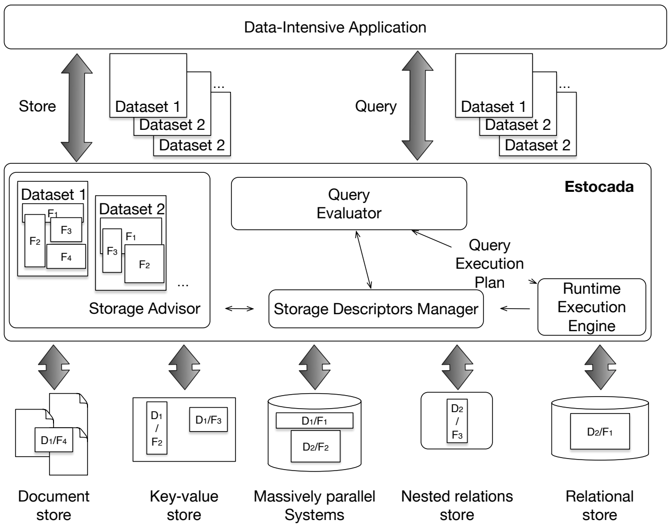


**Figure 1: Estocada architecture.**

**Architecture** Figure 1 outlines the architecture of our prototype based on the above discussion. We assume the typical application uses many data sets $D_1, D_2, \ldots, D_n$, 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 Descriptor Manager* stores information about the available data fragments $D_1/F_1$, $D_1/F_2, \ldots, D_1/F_n$, $D_2/F_1, \ldots$ etc., and where they are stored in the underlying DMSs, illustrated by a NoSQL store, a key-value store, a document store, one for nested relations, and finally a relational one. 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, 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 Advisor* recommends dropping redundant fragments that are rarely used or under-performing, and adding new fragments that fit recently heavy-hitting queries. To solve this problem across data models, we once again exploit our pivot model to reduce to the novel setting of relational view selection *under constraints*.

The *Query Evaluator* receives application queries. If a query carries over a single source $D_i$, the query will likely be in the native language of $D_i$. If the query carries over multiple sources having different data models, this assumes the existence of a global-as-view integration layer on top of the (application-transparent) local-as-view approach internally followed by ESTOCADA. While we do not focus on this (optional) GAV integration layer, in such a case we assume the query is specified by combining algebraic operations (such as filter, join, union, etc.) on top of individual queries carrying over each dataset. It is rather straightforward then to translate such a query in the pivot model, by focusing first on the queries confined to a data source, and then on the combination operators. The evaluator looks up the storage descriptors corresponding to fragments of the queried datasets, calls the PACB engine to obtain rewritings. The *Runtime Execution Engine* then translates such rewritings into excutable ones as described above and evaluates them.

Clearly 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 combination of available systems.

# 4. DEMONSTRATION OUTLINE

We will show ESTOCADA in action on a set of scenarios closely derived from the one described in Section 2, on datasets obtained through the Big Data Benchmark [4], and server logs from several actual e-commerce platforms to which we gained access through the Datalyse project. To illustrate the scenarios we will use Postgres, Redis, and Spark as the underlying storage systems.

The demo attendee experience is as follows: **1.** Pick a dataset, which comes with a previously specified workload and a set of possible fragments; chose a subset of the fragments, view their specification in the source native language, and after translation to the pivot internal model. **2.** Pick a workload query and trigger its rewriting: inspect its translation in the pivot model, the output of the PACB rewriting algorithm, its translated form and finally the executable plan. **3.** Trigger the execution of the rewriting, which outputs a set of performance statistics split across the underlying DMS and ESTOCADA' runtime. We will provide for each dataset the specification of one fragment which stores it "as such" in a DMS of its native data model; this will enable comparing performance between the vanilla (one-store) execution and the one enabled by multiple stores.

# 5. RELATED WORK AND CONCLUSION

Heterogeneous data integration is an old topic [8,13,20,22] but the remark "one-size does not fit all" [24] has been recently revisited [16, 21]. The performance benefits of using multiple stores together (a Hadoop one and a relational database) have been demonstrated in [19]; they select relational views to be materialized based on cost information, but do not handle multiple data models through a unified approach as we do. Polystores [9, 10] 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. This contrasts with our LAV approach where the data store variety is hidden to the application layer. The integration of "NoSQL" stores has been considered e.g., in [3] again in a top-down GAV approach without considering materialized views.

Adaptive stores for a single data model have been studied e.g., in [2, 7, 14, 17, 18]; views have been also used in [1, 23] 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 [11, 12] allow migrating data between two different schemas. We aim at providing to the applications transparent data access to heterogeneous systems, relying on fundamentally different rewriting techniques.

View-based rewriting and view selection are grounded in the seminal works [13, 20]; 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 [8, 22] but ESTOCADA generalizes these works by allowing multiple data models both at the application and storage level.

To conclude, we believe hybrid (multi-store) architectures have the potential to bring huge performance improvements, since (redundant) views storing query results can increase the efficiency of query evaluation by many orders of magnitude. ESTOCADA [5, 6] supports this by a local-as-view approach whose immediate benefit is flexibility since it requires no work when the underlying data storage changes; we demonstrate its performance benefits and the interest of simple storage recommendation heuristics. Our work is ongoing toward a cost-based recommendation of optimal fragmentation.

# 6. 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] Big Data Benchmark. `https://amplab.cs.berkeley.edu/benchmark/`.

[5] F. Bugiotti, D. Bursztyn, A. Deutsch, I. Ileana, and I. Manolescu. Invisible Glue: Scalable Self-Tunning Multi-Stores. In *CIDR*, 2015.

[6] F. Bugiotti, D. Bursztyn, A. Deutsch, I. Manolescu, and S. Zampetakis. Flexible hybrid stores: Constraint-based rewriting to the rescue. In *ICDE*, pages 1394–1397, 2016.

[7] D. Dash, N. Polyzotis, and A. Ailamaki. Cophy: A scalable, portable, and interactive index advisor for large workloads. *PVLDB*, 4(6), 2011.

[8] A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *VLDB*, 2003.

[9] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden,

D. Maier, T. Mattson, and S. B. Zdonik. The BigDAWG polystore system. *SIGMOD Record*, 2015.

[10] A. Elmore, J. Duggan, M. Stonebraker, and al. A Demonstration of the BigDAWG Polystore System. *PVLDB*, 2015.

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

[12] 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.

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

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

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

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

[17] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-in-time data virtualization: Lightweight data management with ViDa. In *CIDR*, 2015.

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

[19] J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, and al. MISO: souping up big data query processing with a multistore system. In *SIGMOD*, 2014.

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

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

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

[23] J. Shute, R. Vingralek, B. Samwel, and al. F1: A Distributed SQL Database That Scales. In *PVLDB*, 2013.

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