A First Step Towards Integration Independence

Laura M. Haas [1], Renée J. Miller [2], Donald Kossmann [3], Martin Hentschel [3]

[1]*IBM Almaden Research Center, USA*
laura@almaden.ibm.com
[2]*University of Toronto, Canada*
miller@cs.toronto.edu
[3]*ETH Zurich, Switzerland*
{donald.kossmann, martin.hentschel}@inf.ethz.ch

***Abstract*—Two major forms of information integration, federation and materialization, continue to dominate the market, embedded in separate products, each with their strengths and weaknesses. Application developers must make difficult choices among techniques and products, choices that are hard to change later. We propose a new design principle, *Integration Independence*, for integration engines. Integration independence frees the application designer from deciding how to integrate data. We then describe a new, adaptive information integration engine that provides the ability to index base data or to materialize transformed data, giving us a flexible platform for experimentation.**

I. Introduction

Information integration is a ubiquitous and expensive challenge. Businesses and scientists alike struggle with it. Decades of research have brought progress, in the form of engines and tools that, when applicable, make integration significantly easier [1]. However, the proliferation of engines and tools has created more confusion, and may even be lengthening the time needed to provide integrated information for a new application [2].

Information integration engines typically fall into one of two broad classes. *Federation engines* (*aka* mediators [3]) provide a virtual view of underlying data, as if it were already integrated and transformed, and in a single source [4], [5], [6]. This virtual view describes a plan for integrating information and is designed *a priori*. The plan cannot be specific to a single database state, but must work even as the data changes over time. When a query is submitted by an application, the virtual view is used to find and transform the required information, a *lazy* form of integration. (Peer-to-peer systems [7], [8] and agent-based systems [9] also implement a lazy approach to integration; we focus on federation below, since that is the basis for most commercial tools for lazy integration). *Extract, Transform, Load (ETL)* tools [10], [11], by contrast, *eagerly* integrate data, producing a materialized collection of results in advance. When a query comes in, it is addressed to and answered by the materialized integrated data.

Each of these techniques has its merits, and its issues. Lazy integration only integrates the data that is needed, so if only a small portion of the various sources will ever be touched, it uses less time and resources. Likewise, if the underlying data in the various sources changes frequently, it saves the work of constantly re-building the materialized view. However, if the data sizes are massive, if the data from the various sources has conflicts that must be reconciled, if the queries are too complex and would require large volumes of data to be moved in order to compare records, and so on, then eager integration is a better choice. It gets the hard work out of the way "once and for all" if the data is reasonably static, and the results can be materialized locally in a form that makes the expected queries easy to answer. Often, lazy solutions are easier to set up; eager integration requires planning and painstaking design up front. Typical warehouse projects may take six months to a year or more before they are ready to use; a federation can produce results in as little as a few days or weeks.

Recently, research in information integration has focused increasingly on design-time issues: how to do matching and mapping, what are the right algorithms for entity resolution, how to cleanse the data, what degree of semantic integration will be supported and so on. These are certainly important problems. However, what is arguably the most challenging design-time problem, namely, choosing an integration engine, has gone, for the most part, unaddressed. The choice of an integration engine often determines what design-time tools are appropriate and can be used. For example, most data cleansing tools are batch-oriented, appropriate for use with an ETL solution, but not with federation or other lazy approaches. As input, ETL engines take a script, typically generated by a design tool that works with a data flow paradigm. By contrast, lazy engines typically expect a query in whatever language they support, so different tools such as SQL query builders are needed. Few tools can generate instructions for both lazy and eager engines. (One exception is Clio [12], [13], which can generate SQL, SQL/XML, XPath, and ETL scripts as needed). Hence, an integration engine must be chosen early in the design process, and changing engines later is difficult.

Recent work on runtimes for integration has looked at integration in a variety of environments (for example, the deep web [14], or personal information spaces [15]), creating new engines for specific tasks. Exploring one emerging context, in which there may be thousands of schemas (*extreme schema heterogeneity*) and many different desired views of data, led to a new integration technique known as *Mapping Data to Queries (MDQ)* [16]. This method of integration is similar in spirit to that of adaptive query processing techniques [17], [18] in standard query processing. Such techniques allow a query processor to adapt its query processing (e.g., join order) to the actual data, which may have different characteristics than those predicted when the query was statically optimized. Similarly, MDQ allows the integration engine to continually adapt the

integration to the data that is actually seen, applying only those mapping rules needed for the data instead of rewriting the query to cover all possible schemas and mapping rules. MDQ is actually even lazier than federation, which typically applies mappings at compile-time of the query, whereas MDQ applies them during query execution itself. MDQ therefore expands the spectrum of integration possibilities, from eager (materialize the result during design-time) to lazy (plan how to retrieve the result at query compile-time) to adaptive just-in-time (react to the data during query execution itself).

We believe that it is time to revisit integration runtimes. We propose a new principle, *Integration Independence* [19], for guiding the design of integration engines. Integration independence is the idea that applications should be immune to changes in how and when information is integrated. Whether the data asked for in a query is already integrated or will be integrated as needed should be transparent to the application. We envision a new type of engine for running applications that can easily move between just-in-time, lazy and eager based on the data and the workload. This enables the style of integration to be viewed as an optimization choice, and opens several interesting research directions. If how we integrate is simply an optimization, then we should have choices in the middle, between eager and just-in-time, to meet the needs of an application. What are these new execution methods? How can we design an integration engine to exploit them? How will we optimize an application in this new world, so that the best integration methods are used? What is the right unit of optimization: a single query, an application, or a workload?

Integration independence also suggests new performance metrics. With integration independence as a guiding principle, we can talk about measuring the "overhead" of an integration, where the overhead is defined as the difference in response time or throughput of executing a workload against the integration, versus executing it over a fully clean and integrated source. This is in contrast to design-time work which has focused on measuring the effort required to write an integration plan. Here we are concerned with the overhead an application sees, the cost at runtime of a query. Of course, other metrics such as update cost and overhead will also be required.

We are building a system that covers the full spectrum of integration styles, combining the adaptive MDQ technique, indexes and materialized views. We intend to use this system for experimentation, with an eventual goal of achieving true integration independence. In the next section, we describe MDQ and the architecture of our system.

II. A New Integration Engine

Our new engine can handle a spectrum of integration scenarios. At one extreme, the data may be fully distributed, with no materialization. When a query arrives, the data is processed using the MDQ technique to filter out data that will not match the query given the current set of mapping rules. As usage patterns are established, some data may be indexed, or even transformed and materialized. At the other extreme, all transformations needed for a given set of applications are done in advance, and the results materialized, emulating a warehouse. We intend to experiment with this engine to see how to support true integration independence.

Note that we could do something similar with a sophisticated federation engine today. By leveraging the materialized view facilities, and/or caching capabilities that many of these systems include, an application designer or database administrator could simulate at least a portion of this range. However, limitations in existing caching and materialized view facilities restrict their applicability. Also, no existing federation engine can offer the extreme, adaptive just-in-time integration offered by MDQ. MDQ indexes mapping rules and applies only the rules that are needed at runtime as data is processed. In [16] we demonstrate the performance advantages that this adaptive just-in-time integration offers over the standard query rewriting techniques employed by federation engines. Thus, we decided to build a new engine for our experiments.

Our prototype makes some simplifying assumptions. For the moment, we look only at XML data. XML is a flexible data description format, and almost any source's data can be represented as XML. We further assume that all the data to be integrated is stored in a file system, one document per file. (Data can also be fetched from the web or other sources, and stored in memory for processing). This allows us to ignore the well-understood difficulties of connecting to different types of data sources and to focus on our integration engine.

In the following, we first describe the MDQ technique in more depth. Then in Section II-B we give an example of how performance issues can arise, how they are addressed in today's systems, in our new system, and in the ideal, integration independent system.

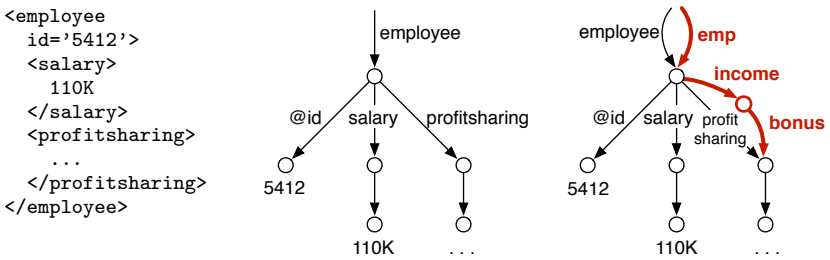
A. Mapping Data to Queries

In [16], we describe a fundamentally new approach to query processing and data integration with mapping rules. This technique scans the data, applying mapping rules that are relevant to the query during the query processing itself to decide whether a document should be returned. The mapping rules are indexed for fast access. By contrast, typical federation systems rewrite a query to take into account all possibly relevant mapping rules, before beginning query execution, while ETL systems apply all mapping rules before the query is even submitted, materializing the results. Since we map the data in response to the query, rather than rewriting the query, we call this method *mapping data to queries (MDQ)*.

We support mapping rules of varying degrees of complexity, from simple *is-a* relationships to the ability to restructure data and to compute complex functions. Each rule relates a query over a source to a query over a target, using the following syntax:

source-expr [**as** $*variable*] → *target-expr*

In this syntax, **as** and → are keywords. The expressions *source-expr* and *target-expr* can be any XPath expressions [20] with forward axis steps. Furthermore, the target expression can be an XQuery element constructor. As an example, the simple rule *employee* → *emp* says that an *employee* element can be



(a) XML Snippet (b) Tree (c) Annotated Tree

Fig. 1: Annotating Data with MDQ

returned whenever a query asks for an *emp*. Examples of more powerful rules can be found in [16].

MDQ annotates the internal representation of the data at runtime, to reflect how mapping rules affect parts of the document. Figure 1 illustrates how MDQ annotates data with an example. Figure 1a shows a small XML snippet. Figure 1b shows how this XML snippet is represented as a tree. The labels of edges represent the names of attributes or elements in the original XML snippet. Leaf nodes contain values (e.g., the string "5412") of the original XML snippet. Figure 1c shows the annotated tree after applying two mapping rules. The rule *employee* → *emp* adds an additional edge into the tree to reflect that the root node also matches a query which asks for an *emp*. The second rule, *profitsharing* → *income/bonus*, adds an additional node and two edges to show that *profitsharing* can be reached by the path expression *//income/bonus*.

We only augment those parts of a document that are relevant to the query. If the whole document is irrelevant, then the mapping rules will not be executed and no data will be annotated. In order to scale with the number of mapping rules, MDQ indexes the source expressions of mapping rules so that mapping rules that match part of a document can be found quickly at runtime.

The difference between MDQ and any other data integration system is that the processing of mapping rules forms a virtual data layer (the annotations). We are able to produce (portions of) this layer just-in-time. Of course, we only generate the relevant portions taking into account both the data and the queries. Because we index the mapping rules, this just-in-time generation is very fast. We also have the option of persisting portions of this virtual layer to speed up query evaluation (i.e., caching annotations and writing them to disk). If the database administrator chooses to add indexes, these indexes will index right into the virtual layer. On the other hand, materialized views will produce a cleaned up version of the data (a new document) without any additional annotations. We are then back to normal query processing, but with a document that is more aligned with the queries.

The big advantage of the MDQ approach is that it scales well with the number of mapping rules. MDQ also performs better than query rewrite if the mapping rules are complex or if the queries are complex and involve many path expressions. Furthermore, it is possible to add (and remove) mapping rules without recompiling any queries. In this way, MDQ implements a common design-time principle known as pay-as-you-go [21]. As more rules, providing more sophisticated semantic integration become available, MDQ can automatically take advantage of them. Hence, MDQ can accommodate settings with a large and dynamic set of mapping rules.

B. Where Integration Independence Would Help

Consider an application running on a company's HR database. The application routinely issues queries to monitor the company payroll, and to generate reports for each department. The company is doing well and over a period of time purchases a dozen smaller companies. A federated approach is chosen to integrate the HR systems of these companies. Integrated views are designed. Suddenly, reports that took seconds to build are taking hours. While the amount of data used by the application has increased, perhaps doubling in size, this increase does not account for the terrible performance. The true culprit is the query complexity that results from using the integrated views.

As an example, consider a query to list all employee payments per department for the standard monthly report. When the query was originally written, it "joined" the *emp* collection to the *dept* collection on department number. As new acquisitions were merged in, union views were created over all the collections that map to *emp* and *dept* (possibly with other transformations). So what looks like a simple join is by now really a complex join of unions. When a new acquisition is made, the views have to be redefined; when the query is executed, it now is more complex. The query optimizer has to decide whether to first create the unions, and then join the results, or whether to transform the join of unions into a union of joins. Few optimizers do a good job on these queries; often there is insufficient information about the data for the optimizer to decide which approach is best.

The poor Database Administrator (DBA) who is called on to deal with this awful performance must create copies of portions of the data, for example, to materialize the department view. However, depending on the view query, it may not be possible to use the view without changing the query to directly use the stored version, or it may not be possible to keep the materialized copy up to date automatically. (In our scenario, a

materialized view of employees or departments can probably be maintained automatically. A materialized join, however, might be harder, depending on the predicates involved). The DBA will end up with a maintenance headache. There are few tools to help him decide what (and how much) to materialize.

How would things work with our new engine? The primary difference is that the mappings are not compiled ahead of time into view definitions. Instead, as the query starts executing, data items are retrieved, the mapping rules that might apply (based on the query) are compared to the data, and the data is annotated (if it hasn't been previously) as appropriate. Query execution then proceeds as usual. So the major difference is that the problem of union views is finessed, and performance probably does not degrade so badly.

Still, we now have much more data to look at – so likely we will want to add some form of materialization, whether by indexing the data based on structure (including annotations), or by materializing (portions of) the query result. In our system this works exactly as in today's database management systems. When we construct an index, we specify the domain that needs to be indexed and the indexed properties (e.g., the salary of all employees). While building the index, MDQ takes effect and integrates the data, applying the mapping rules while the data is scanned. In order to decide whether an index (or materialized view) is applicable to a query, query/view subsumption needs to be applied, just as in any database system that make use of indexes and materialized views. Over time, our system moves from a federated style of data integration to a more data warehouse like style. The DBA has less headaches on what data to integrate using which technology.

In the ideal integration independent system, the materialization of data would not force a change in the queries. Now that we have a single system that can span the range from fully materialized to fully dynamic, we intend to experiment with the engine to find how best to leverage these capabilities. We hope to find "integration plans" that give us flexibility to meet performance goals without changing the queries, thus achieving true integration independence.

III. Future work

The principle of integration independence will be critical for the next generation of industrial integration engines. Once the key properties of an "integration independent" engine are understood, we should be able to build such systems for a broad variety of design points and architectures. In the immediate future, though, our goal is to demonstrate a working engine in our current, idealized context, and from there, to understand and characterize its performance. We hope that at an extreme of materialization it will show little overhead compared to a fully integrated dataset, i.e., performance comparable to a prebuilt warehouse. At a "just-in-time" extreme, we hope it will do as well or better than "real" federation. And we expect it to move seamlessly between the two extremes. Once it can do that, we will look at how to exploit these features for true integration independence.

In [19], we first broached the idea of integration independence, along with another principle we call *holistic integration*. Holistic integration recognizes the dependency between runtime and design-time, and the need to move flexibly between the two. To facilitate the often iterative process of integration design, and support the virtuous cycle that arises, holistic integration attempts to do many or most of the design-time tasks in the same framework as the runtime. We would like, in the further future, to explore the use of our new engine to support a holistic integration environment.

References

[1] P. A. Bernstein and L. M. Haas, "Information integration in the enterprise," *Communications of the ACM*, vol. 51, no. 9, pp. 72–79, 2008.

[2] L. M. Haas, "Beauty and the Beast: The Theory and Practice of Information Integration," *LNCS*, vol. 4353, pp. 28–43, 2007, ICDT.

[3] G. Wiederhold, "Mediators in the Architecture of Future Information Systems," *IEEE Computer*, vol. 25, no. 3, pp. 38–49, 1992.

[4] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang, "Optimizing Queries across Diverse Data Sources," in *VLDB*, 1997, pp. 276–285.

[5] A. Y. Levy, A. Rajaraman, and J. J. Ordille, "Querying Heterogeneous Information Sources Using Source Descriptions," in *VLDB*, 1996, pp. 251–262.

[6] S. Chawathe *et al.*, "The TSIMMIS Project: Integration of Heterogeneous Information Sources," in *Proc. of the 100th Anniversary Meeting of the Information Processing Society of Japan(IPSJ)*, Tokyo, Japan, Oct. 1994, pp. 7–18.

[7] Z. G. Ives, T. J. Green, G. Karvounarakis, N. E. Taylor, V. Tannen, P. P. Talukdar, M. Jacob, and F. Pereira, "The ORCHESTRA Collaborative Data Sharing System," *SIGMOD Record*, vol. 37, no. 3, pp. 26–32, 2008.

[8] P. Rodríguez-Gianolli, M. Garzetti, L. Jiang, A. Kementsietsidis, I. Kiringa, M. Masud, R. J. Miller, and J. Mylopoulos, "Data Sharing in the Hyperion Peer Database Systens, *A System Demonstration*," in *VLDB*, 2005, pp. 1291–1294.

[9] M. H. Nodine, A. H. H. Ngu, A. R. Cassandra, and W. Bohrer, "Scalable Semantic Brokering over Dynamic Heterogeneous Data Sources in InfoSleuth™," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 5, pp. 1082–1098, 2003.

[10] IBM Corp., "InfoSphere DataStage," http://www-01.ibm.com/software/data/infosphere/datastage/.

[11] Informatica, Inc., "Data Warehousing," http://www.informatica.com/solutions/enterprise_data_warehouse/Pages/index.aspx.

[12] R. J. Miller, L. M. Haas, and M. Hernández, "Schema Mapping as Query Discovery," in *VLDB*, 2000, pp. 77–88.

[13] R. Fagin, L. M. Haas, M. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis, "Clio: Schema Mapping Creation and Data Exchange," in *Conceptual Modeling: Foundations and Applications, Essays in Honor of John Mylopoulos*. Springer, 2009, vol. 5600, pp. 198–236.

[14] G. Kabra, Z. Zhang, and K. C.-C. Chang, "Dewex: An Exploration Facility for Enabling the Deep Web Integration," in *IEEE ICDE*, 2007, pp. 1511–1512.

[15] Y. Cai, X. L. Dong, A. Y. Halevy, J. M. Liu, and J. Madhavan, "Personal information management with SEMEX," in *ACM SIGMOD Conf.*, 2005, pp. 921–923.

[16] M. Hentschel, D. Kossmann, D. Florescu, L. Haas, T. Kraska, and R. J. Miller, "Scalable Data Integration by Mapping Data to Queries," ETH Zurich, Dept. of Computer Science, Tech. Rep. 633, 2009.

[17] R. Avnur and J. M. Hellerstein, "Eddies: Continuously Adaptive Query Processing," in *ACM SIGMOD Conf.*, 2000, pp. 261–272.

[18] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld, "An adaptive query execution system for data integration," in *ACM SIGMOD Conf.*, 1999, pp. 299–310.

[19] L. M. Haas, M. Hentschel, D. Kossmann, and R. J. Miller, "Schema AND Data: A Holistic Approach to Mapping, Resolution and Fusion in Information Integration," in *Int'l Conf. on Conceptual Modeling (ER)*, 2009, pp. 27–40.

[20] J. Clark and S. DeRose, "XPath 1.0," http://www.w3.org/TR/xpath.

[21] M. J. Franklin, A. Y. Halevy, and D. Maier, "From databases to dataspaces: a new abstraction for information management," *SIGMOD Record*, vol. 34, no. 4, pp. 27–33, 2005.