Polystore Query Rewriting: The Challenges of Variety

Yannis Papakonstantinou*yannis@cs.ucsd.edu

Numerous databases marketed as SQL-on-Hadoop, NewSQL [16] and NoSQL have emerged to catalyze Big Data applications. These databases generally support the 3Vs [7]. (i) Volume: amount of data (ii) Velocity: speed of data in and out (iii) Variety: semi-structured and heterogeneous data. As a result of differing use cases and design considerations around the Variety requirement, these new databases have adopted semi-structured data models that vary among each other. Their query languages have even more variations. Some variations are due to superficial syntactic differences. Some variations arise from the data model differences. Other variations are genuine differences in query capabilities. Yet another kind of variations involves subtly different semantics for seemingly similar query functionalities. E.g., equality may have subtle and unexpected meanings in the presence of missing attributes in NoSQL databases.

Even in a single organization, it is common to find multiple databases that exhibit high variety. Often applications require integrated access to those databases. It is difficult to write optimized software that retrieves data from multiple such databases, given the different data models, different query syntaxes and the (often subtly) different query semantics. This problem has been recognized for many decades in the database community. It is now accentuated, as a plethora of different and specialized databases finds its place in the enterprise. For example, the problem happens whenever an enterprise adopts a fast and scalable NoSQL database to capture its users' activity on its web site (web log) and then builds applications that need integrated access to the web log data stored in the NoSQL database and also to data in its existing SQL databases.

1. REWRITING

Mediator systems had been proposed in the 90s in order to provide integrated query access to multiple heterogeneous databases, including databases with different query capabilities. Polystores provide similar functionality [6]. UCSD's FORWARD Middleware is a mediator. Figure 1 shows an example of how the FORWARD middleware evaluates queries over different databases with varying capabilities. An organization owns a PostgreSQL database and a MongoDB database, where the PostgreSQL database contains a `sensors` table and MongoDB contains a `measurements` array of JSON objects. Conceptually, the FORWARD Middleware presents to its clients the virtual SQL++ views **V1** and **V2** of these databases. (We describe the SQL-backwards-compatible SQL++ below.) The **`sensors`** table of the Postgres database is presented in the view **V1** as a bag (bags are denoted by **{{...}}**) of JSON objects, since both tuples and JSON objects are sets of attribute/value pairs. Notice, the virtual view **V2** of MongoDB is identical to its native data representation. Since the views are virtual, the FORWARD Middleware does not have a copy of the source data.

Suppose the client issues the federated query **Q**, which finds the average temperature reported by any reliably functioning sensor in a specific lat-long bounding box, where a sensor is deemed reliable only if none of its measurements are outside the range $-40°$F to $140°$F. The query can be rewritten into the following plan, which includes PostgreSQL and MongoDB subqueries that are efficient and compatible with the limited query capabilities of MongoDB.[1]

Plan 1: $(q_{\mathbf{Q1}} \overset{s.id \rightarrow @id}{\ltimes} p_{\mathbf{Q2}}) \overset{s.id \rightarrow @id}{\bowtie} p_{\mathbf{Q3}}$

The plan first issues to PostgreSQL the query **Q1**, which finds the ids (**`s.id`**) of the sensors in the bounding box. Then, for each id, the parameter-passing semijoin operator issues to MongoDB repeated queries **Q2**, each time instantiating the parameter `@id` of **Q2** with another id from the result of **Q1**. The queries **Q2** test whether the identified sensor is reliable. If the sensor qualifies, the particular sensor from **Q1** appears in the output of the semijoin. Finally, the parameter-passing join issues to MongoDB repeated queries **Q3** that find the average of the temperature measurements for each qualified sensor. *Notice that if MongoDB had the capability to support nested queries, it would have been possible to issue a single MongoDB query for each id as opposed to two queries.* Therefore the rewriting is aware of source capabilities. Finally, the `coord_to_state()` function, which inputs coordinates and outputs the name of the corresponding state, is executed in the middleware level.

Plan 1 is one of the many possible plans that one may consider. For example, another plan, say Plan 2, could have issued a query **Q1'** that fetches to the middleware all the sensors, regardless of whether they are in the bounding box

*Supported by NSF DC 0910820, NSF III 1018961, NSF IIS 1237174, Informatica and Couchbase grants.

[1] In the interest of simplicity we simplify some aspects of the plan.

or not. Consequently, the bounding box selection would happen at the middleware level and then Plan 2 could proceed with the parameter-passing semijoin and join of Plan 1. However, one can easily tell that Plan 2 is not competitive since the capabilities of Postgres guarantee that one is always better off executing this kind of selection conditions at the Postgres source. Even in the absence of statistics and a cost estimator, a mediator should characterize Plan 2 as non-competitive and worthless of further consideration.

Nevertheless, in general, there are multiple competitive plans, i.e., plans that one needs source statistics and a cost estimator to tell which one is the expected best. For example, if it were the case that the vast majority of sensors are unreliable (i.e., if only a few sensors passed the conditions > -40 and < 140), then it could made sense to execute a Plan 3 that first issues a MongoDB query to find the unreliable sensors, then checks on Postgres whether they are in the wanted bounding box and then finds the average tempratures. One needs statistics and a cost estimator to tell which one of Plan 1 or Plan 3 is better. Therefore they are both competitive.

We define the rewriting problem as follows: *Given a client query* $\boldsymbol{Q}$*, find all competitive candidate plans that are non-trivially different.* By requiring non-trivial difference, we eliminate the possibility of having multiple candidates that are superficial variations of each other. For example, superficially different plans can be created by, say, creating two versions of a plan, where the two versions use different, yet result-equivalent and performance-equivalent source queries.

In this paper we assume that the optimal plan is found in a sequence of two modules. The first module is the rewriter: It inputs the client query and it outputs candidate competitive rewritings. The second module is the cost optimizer: It inputs the competitive candidate rewritings, assigns a cost to each one of them and chooses the plan with the minimum cost. In contrast, many query processors generally mix rewriting and cost optimization, hence pruning the search space and/or efficiently exploring the space. (By "space" we refer to the space of candidate rewritings.) The simpler, modular architecture of this paper splits the rewriter from the cost optimizer, primarily for presentation reasons: We show that rewriting poses challenging problems, even if not mixed with cost estimation and pruning.[2]

The rewriting problem is now more relevant and more challenging than it was in the past, since we now face unprecented variety in the capabilities of sources. This paper presents the past techniques and architectures for query rewriting, argues that continued work is needed in order to become practically viable in the present environment and provides corresponding directions. Notice that "practical viability" means being able to develop mediators where the number of lines of code (LOC) grows ideally sublinearly or, in the worst case, linearly in the number of involved sources. Given the number of possible different sources, any superlinear growth of LOC all but guarantees that such system will not become a general purpose mediator, but rather will become federators specializing to a few particular sources.

[2]Furthermore, it is an open question, to be judged in real-world deployments, whether pruning of the search space is a significant feature in mediators and polystores. In relational database optimizers, where rewriting, estimation and pruning are mixed, n-way joins effectively prohibited architectures where all equivalent plans (join orders) would be explicit produced, as their number would generally be exponential in n. In contrast, the number of competitive plans may rarely be huge, therefore demoting the importance of mixing rewriting and optimization.

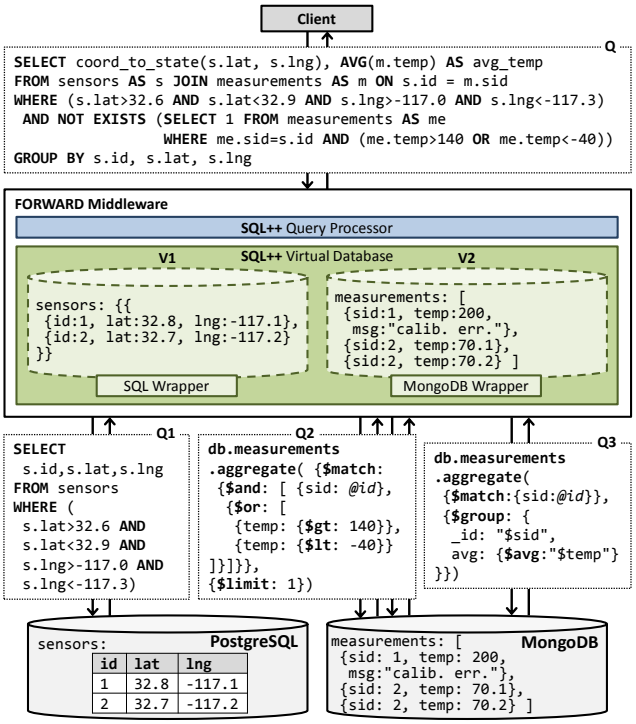

Figure 1: FORWARD Query Processing Example

2. DESCRIBING SOURCE CAPABILITIES

The prior example made clear that knowledge of the query capabilities of the sources is essential in the rewriting problem. To further isolate the point that requires capabilities awareness, during the rewriting process, we further modularize the rewriting module into two sub-modules. The split decouples rewriting aspects that pertain to fundamental capability differences (e.g., can a source do nested conditions or not?) from rewriting aspects that are a translation across superficial syntactic differences (e.g., a source can do nested conditions but it expresses them in its own, non-SQL dialect).

Architecturally, we enable this split by formally specifying the syntax and semantics of SQL++ [12], which is a unifying semi-structured data model and query language that is designed to encompass the data model and query language capabilities of NoSQL, NewSQL and SQL-on-Hadoop databases. The reader is referred to [11] for its full details.[3] The virtual views shown in Figure 1 follow the SQL++ data model. Furthermore, we assume that each virtual view can be queried with SQL++. However, in light of the limited ca-

[3] The SQL++ semantics stands on the shoulders of the extensive past work from the database R&D community in non-relational data models and query languages: OQL [2], the nested relational model and query languages [9, 15, 1] and XQuery (and other XML-based query languages) [14, 5, 4]. SQL++ is an extension to SQL and is backwards-compatible with SQL. This choice was made in order to facilitate the SQL-aware audience. However, choosing SQL compatibility is not an important property of the rewriting problem or of mediators. The issues discussed here will emerge even using a different, non-SQL-compatible language.

pabilities of the sources, the more precise statement is that each virtual view can answer a subset of SQL++ queries, according to the source capabilities. For example, the virtual view of MongoDB can be queried with SQL++ but such SQL++ cannot have nested conditions. The important differentiating aspect of SQL++ is that its syntax includes *configuration flags*. For each database system, we set the flags to different values, typically to describe whether the database system supports the corresponding part of the language. For example, a configuration flag represents whether the nested conditions part of the syntax is allowed. When we describe a Postgres database this flag will be true. In a MongoDB it will be false. While [12] describes the flags by laying them on the SQL syntax, the actual FORWARD uses the respective flags on the algebra level.

There are also additional flags that describe subtle semantic differences. For example, equality in SQL and equality in MongoDB have subtly different semantics in the case of nulls and absent attributes. Appropriate configuration flags classify these differences among the sources. In the case of the equality problem, appropriate rewriting rules will then translate an equality under SQL's configuraton flag, into an equality under MongoDB's flag.

By appropriate choices of configuration options, the SQL++ semantics morph into the semantics of other SQL+JSON databases. The short paper [12] shows how the SQL standard and four well known (Cassandra CQL, MongoDB, Couchbase N1QL, AsterixDB AQL) semi-structured database query languages, are explained as particular settings of the configuration options. While SQL++ does not support the exact syntax of any of these four databases, it can be morphed by the configuration options to support equivalent queries.[4] To further facilitate the readers' understanding of SQL++ and the effect of the various configuration options, we provide a web-accessible reference implementation of SQL++ at `http://forward.ucsd.edu/sqlpp`.

3. NAIVE QUERY PLAN DECOMPOSITION WILL NOT WORK

A simple and easy-to-implement algebra-based architecture adds the capabilities-based rewriting as a very last step in otherwise conventional rewriting [8]. In our running example, this would mean that the rewriter first employs the usual palette of beneficial rewritings, such as pushing conditions down, in order to produce an optimized algebraic expression that is still *source-agnostic*, in the sense that it does not specify what SQL++ queries must be sent to the SQL++ views of the underlying sources. Then a *query decomposer*, which is aware of the query capabilities, inspects the source-agnostic plan and finds its biggest subexpressions that can be executed in a single source. Then it formulates corresponding SQL++ queries.

Such architecture is a relatively obvious and simple incremental improvement over conventional query processors. The bad news is that in the era of high variety there cannot be a uniform set of conventions for the format of the source-agnostic plans. Ideally the source agnostic plans would have a format that would allow decomposers to always find the optimal subexpressions/subqueries. Such ideal source agnostic plans are generally impossible in the presence of high variety.

We exemplify next a very common, albeit not the only one, problem pattern that argues for the non-existence of "ideal" source agnostic plans. Consider two algebraic operators f and g that can be swapped, i.e., $f(g(.) = g(f(.)$. Assume that the source-agnostic plan for a client query q is
Expession A1: $o_1(f(g(e_2)))$

By virtue of the equivalence of $f(g(.))$ and $g(f(.))$, one can see that the rewriter could have also produced the source-agnostic Expession A2 $o_1(g(f(e_2)))$ instead of Expession A1. In the absence of knowledge of the capabilities of the unerlying sources, it is impossible to choose between A1 and A2. The inability to choose can eventually lead to suboptimal plans, because the decomposers will detect suboptimal subexpressions. For example, in the discussed problem pattern, consider two alternate source capability situations. In both situations e_2 should be executed at a source s, while o_1 can only be executed in the middleware. In the first situation, g can also be executed at the source s, along with o_2, while f cannot be executed at s. In the second situation, it is vice versa: f can be executed at s, along with o_2, while g cannot. It is easy to see that the first situation produces an optimal plan if the source-agnostic plan is the Expession A1. The second situation produces an optimal plan when the source agnostic plan is the Expession A2.

In effect, there is not a one-size-fits-all source agnostic plan. In the absence thereof, one may argue for a seemingly simple fix: First produce *all* possible source agnostic plans. For example, whenever faced with operators that can be swapped produce a plan for each swapping. Then let decomposers work on each source agnostic plan. Unfortunately, this is not a scalable solution since the number of source agnostic plans would be huge.

Instead, FORWARD supports a *capability-aware rewriting* stage that is aware of the capabilities flags and performs rewritings with the goal of maximizing the subexpressions that can be pushed to a source. According to this architecture, it does not matter whether the source agnostic phase produces Expession A1 or Expession A2. Let us, for example, assume that it produces Expession A1, while g cannot be pushed to the source but f can. The capability-aware rewriting phase will inspect operators above e_2 and see if they can be pushed down along with e_2. Since f is such an operator, the rewriting will transform $f(g(e_2))$ into $g(f(e_2))$, hence preparing the ground for the decomposer.

4. MODELING VARIETY BY INFINITELY MANY PARAMETERIZED VIEWS

A number of works tackled the problem of answering queries using sources with limited query capabilities as if it were an extension of the well-studied problem of answering queries using views. Namely, [13, 10, 17, 18] introduced notations that precisely described the (in general, infinite) set of queries that the sources express. All works focused on conjunctive queries, including in semistructured settings as well [13]. At a high level of abstraction, they all enabled the description of the (potentially infinite) set of views by an appropriately expanded notation of Datalog, where the (infinitely many) views correspond to the (infinitely many) views that can be produced by expanding the intensional database predicates of the Datalog program. For example,

[4] An earlier, extended version [11] shows how an additional six databases correspond to particular settings of the configuration options. We expect that some of the results listed in the feature matrices describing configuration options will change in the next years as the space evolves rapidly.

consider the following capabilities description:

```
1: view(doc)        :-  document(id, doc), id=?
2: view(doc)        :-  document(id, doc),
                        conditions(id)
3: conditions(id)   :-  contains(id, w), w=?
4: conditions(id)   :-  conditions(id),
                        contains(id, w), w=?
```

By expanding the **containsconditions(id)** of Line 2 with Line 4 and then with Line 3 we derive that one supported (parameterized) view is

view(doc) :- document(id, doc), contains(id, w1), w1=?, contains(id, w2), w2=?

By replacing the parameters **?** with constants we have conventional views/queries. Other language extensions [18] enable one to describe the capabilities of a source without even knowing its schema. These works often produced *completeness* guarantees, i.e., they would find all competitive candidate rewritings, even under heavy variety.

While theoretically strong, these works did not turn out as-is into industrial processors, despite their authors founding two mediator companies in the early 00s (Nimble, subsequently acquired by Actuate; and Enosys Software, subsequently acquired by BEA Aqualogic). A key reason for their non-adoption is the relatively low variety requirements of the early 00s. Hence it was sufficient to follow variants of Garlic's [8] algebra-based architecture (briefly outlined as the naive approach in Section 3) rather than delving in a radically different query processor architecture. Furthermore, the completeness of the infinite views-based solutions came at the price of focusing on limited query languages (essentially conjunctive), which is a non-starter for a usable processor.

Nevertheless, their ability to describe the sets of supported queries and the fact they offered completeness guarantee, invites drawing ideas from them in the current era of high variability. How can we bring such descriptive power into an algebraic setting?

5. OTHER ISSUES

We do not discuss cost optimization and polystore design [6], while they are also important problems. It is possible that a designer explicitly chose two (or more) different systems based on the application's needs. Furthermore, such designer may have explicitly placed the same data at more than one sources, hence creating rewriting opportunities where one plan may use one source (for such data), while another plan may use another source (for the same data). It is apparent that the presence of the same data set at multiple sources, further increases the rewriting opportunities.

6. REFERENCES

[1] S. Abiteboul, P. C. Fischer, and H.-J. Schek, editors. *Nested Relations and Complex Objects*, volume 361 of *Lecture Notes in Computer Science*. Springer, 1989.

[2] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the O_2 object-oriented database system. In *DBPL*, pages 122–138, 1989.

[3] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. ASTERIX: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[4] A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1):68–79, 2000.

[5] A. Deutsch, M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. A query language for XML. *Computer Networks*, 31(11-16):1155–1169, 1999.

[6] 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*, 44(2):11–16, 2015.

[7] E. Dumbill. What is Big Data?, 2012. `http://strata.oreilly.com/2012/01/what-is-big-data.html`.

[8] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 276–285, 1997.

[9] G. Jaeschke and H.-J. Schek. Remarks on the algebra of non first normal form relations. In *PODS*, pages 124–138. ACM, 1982.

[10] A. Y. Levy, A. Rajaraman, and J. D. Ullman. Answering queries using limited external processors. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, 1996, Montreal, Canada*, pages 227–237, 1996.

[11] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. SQL++: An expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewSQL databases, 2014. `http://db.ucsd.edu/pubsFileFolder/375.pdf`.

[12] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ query language: Configurable, unifying and semi-structured. *CoRR*, abs/1405.3631, 2015. `http://arxiv.org/abs/1405.3631`.

[13] Y. Papakonstantinou, A. Gupta, and L. M. Haas. Capabilities-based query rewriting in mediator systems. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida, USA*, pages 170–181, 1996.

[14] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XQuery 3.0: An XML query language, W3C candidate recommendation, 2013. `http://www.w3.org/TR/2013/PR-xquery-30-20131022/`.

[15] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Trans. Database Syst.*, 13(4):389–417, 1988.

[16] M. Stonebraker. New opportunities for New SQL. *Commun. ACM*, 55(11):10–11, 2012.

[17] V. Vassalos and Y. Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 256–265, 1997.

[18] V. Vassalos and Y. Papakonstantinou. Expressive capabilities description languages and query rewriting algorithms. *J. Log. Program.*, 43(1):75–122, 2000.