BigDAWG Polystore Query Optimization Through Semantic Equivalences

Zuohao She*, Surabhi Ravishankar† and Jennie Duggan‡
Electrical Engineering and Computer Science, Northwestern University
Evanston, IL 60208
Email: *zuohaoshe2013@u.northwestern.edu, †surabhiravishankar2016@u.northwestern.edu, ‡jennie.duggan@northwestern.edu

***Abstract*—A polystore system evaluates queries that span multiple disparate data models; this character introduces a unique query optimization challenge. Specialized database engines such as array and graph databases support partially overlapping sets of query processing operations. Among their common or similar semantics, different systems could have completely different performance profiles for the same query, making their relative usefulness vary from query to query. We hypothesize that a polystore system could exploit this context-dependent disparity of performance by making choices between executing a sub-query locally and migrating the inputs for remote executions. In this work, as part of the larger ISTC BigDAWG project, we examine the challenges of polystore query optimization through the lens of equivalent semantics among back-end databases.**

I. Introduction

The novelty of the polystore database federation architecture is its support of database engines that have distinct data models. The promise of this inclusion allows end-users to simultaneously use multiple query processing engines–such as relational, graph, and array databases –and thus benefit from their relative strengths. To exploit this opportunity, a polystore system requires a smart optimizer that knows *when* to divide the original plan into disjoint sub-queries and *where*, or which engines, to assign these sub-queries to achieve high performance.

Naturally, the "when and where" question comes down to the trade-off between executing a sub-query at the engine where the input is stored and migrating the inputs to a remote engine for accelerated processing. A traditional federated database system typically assumes that all of its sub-systems have similar data models: the engines support mostly the same operators and their efficiencies differ little. Therefore, their optimization strategies primarily relies on minimizing data movement among database instances. In the context of polystore, however, two equivalent complex queries could run at different enough speed on two engines to justify active migrations of intermediate results [2] [3], which makes the trade-off between local and remote query execution non-trivial. Furthermore, engines with different specializations typically support a different set of operators, which implements a still larger set of semantics. The disparities make it difficult to provide complete translation across engines–for instance, the graph database engine Neo4J does not natively support joins; if we want it to perform an equi-join, then we will need to individually adapt the layout of each data set involved.

II. Opportunities in Polystore Query Optimization

In addition to traditionally dominant relational databases such as PostgreSQL, recently there has emerged a plethora of other offerings in the data management world, including array, graph, document-oriented databases. Many of these newcomers acquired their first audiences by filling a specific demand in the data storage and processing community. Designed from the ground up to excel at their specific niche workloads, most of these engines achieve very appealing results within a narrowly-defined scope. It is to be noted, however, many of these engines that stand out at or even dominate one class of workload often do poorly or even fail to support other workloads. Much of these short-comings are due to their implementation-specific limitations.

Take SciDB for an example; it has excellent linear algebra and statistics functionality out of the box and scales out through multidimensional data partitioning. Its built-in support for n-dimensional chunks (SciDB's equivalence of a page), with overlap improves performance on functions such as window aggregates.

While SciDB's linear algebra and statistic operators run fast by themselves, it largely relies on the skills of end-users to construct fast-running queries. SciDB makes a distinction between a dimension (a key) and an attribute (a regular field) and constraints how users could use a fields for an operator depends on whether it is a dimension. This setup limits the potential for incremental tuple-at-a-time optimizations. In addition, SciDB does not support non-integer dimensions. Therefore, while users of a relational database engine could easily group by values along a string column for aggregation, SciDB users have to take the intermediary step to join the table with a separately constructed mapping between the strings and their corresponding integer indexes.

Another example is Neo4J; it is a graph-oriented engine that specializes in queries that operates on nodes and edges of a graph, using its concise and intuitive Cypher Query Language (CQL). To the user, the graph database structures its data as nodes and edges embedded with label and properties. They internally separate the representation of graph data structures

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

and the auxiliary information such as labels to achieve high performances [10].

Neo4J is not suitable for tasks other than simple graph topological analytics. In fact, if a user wants to compute a linear regression among attributes of a set of nodes, then exporting the data for processing elsewhere will be its best option: not only will it be very hard to implement the algorithm in CQL, but, even if a user does manage to form the required queries, the performance will likely make the user regret its efforts.

To illustrate the disparity of performances on specific tasks, we conducted two benchmark experiments featuring fixed-ended path-finding, a common task for graph databases, and sparse matrix multiplication, the building block of linear algebra workloads–and a simple query for array databases. The path-finding benchmark uses a graph data set containing 200,000 nodes and approximately 4 million edges ran on Neo4J 3.0, relational database PostgreSQL 9.4 and array store SciDB 14.12. It requires the engines to list all paths between the ends that take 4 hops or less. The sparse matrix multiplication benchmark uses a square matrix with 200,000 rows ran on PostgreSQL and SciDB. All experiments are sequentially ran on the same machine, and performances are measured in seconds of total query execution time by the Ubuntu system function *time*. We present our results in Figure 1 and Figure 2.

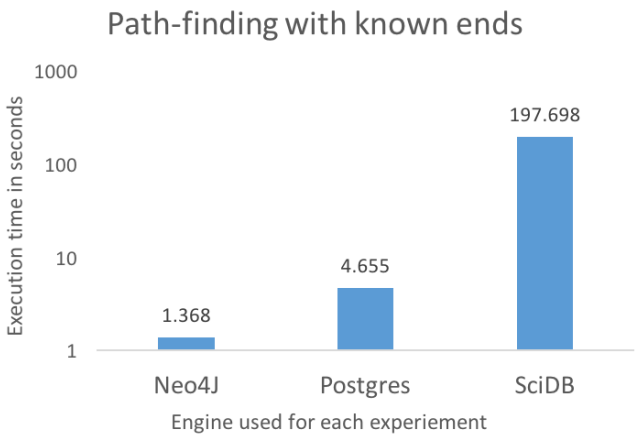


Fig. 1: Performance comparison for a fixed-ends path finding query

Figure 1 shows that Neo4J was 2.5 times faster than PostgreSQL at the path-finding task–repeating the experiment showed similar results. On the other hand, they both outperformed the array database SciDB by a huge margin. SciDB has a specific data partition requirement for any intermediate results; we had to repartition the intermediate results a few times to be allowed to proceed; we suspect that this requirement significantly contributed to SciDB's sluggish performance.

Figure 2 shows the results for the sparse matrix multiplication benchmark. When we observed that SciDB was almost twice as fast as PostgreSQL, we decided to test whether

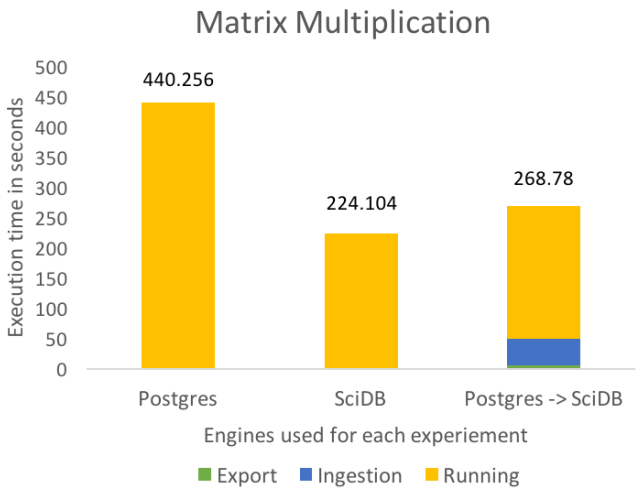


Fig. 2: Performance comparison for a matrix multiplication query

transferring the PostgreSQL data to run on SciDB will be faster than running on PostgreSQL locally. We first copied the PostgreSQL data into a CSV file and then ingested the data to a local SciDB instance to run the query. It turned out that the total time is still shorter than running on PostgreSQL alone by almost 40%. For this benchmark, we did also try to run the query in Neo4J; we omitted this result because Neo4J did not finish the query in ten hours.

It should be noted that when constructing the path-finding queries, Neo4J's CQL enabled us to write very concise graph-oriented queries. While constructing a path-finding query in Neo4J takes 3 simple lines, we had to write 20 for PostgreSQL in SQL and 38 for SciDB in its Array Functional Language (AFL). Constructing the query for SciDB is tedious and error prone and easily gives rise to inefficiencies. It is a work better given to compilers and optimizers.

It is clear that each of the presented niche databases has its relative advantage. To collectively exploit their advantages and circumvent their limitations and pitfalls, from query performances to succinctness of query language, we need to be able to know when and how to rewrite queries in other languages and how to quickly move data around. These are the inspirations for developing the polystore systems.

III. Research Challenges in Polystore Query Optimization

Before delving into polystore query optimizations, we first need to understand the decision space for constructing runnable plans for a given query. As noted in the introduction section, the polystore system determines which engine will process each piece of sub-queries. In light of the rapid emergence of new database systems, semantic properties for specific engines differ significantly. To show this, we listed a few examples of semantic differences among the aforemen-

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

tioned three databases Neo4J, PostgreSQL and SciDB and MongoDB, a popular document-oriented database, in Table I. The differences in data models, query language formats and

	NULL	**Empty**
PostgreSQL	Nulls differ	Whole tuple
SciDB	Nulls identical	"Holes" in arrays
Neo4J	Nulls differ	Similar to PostgreSQL
MongoDB	Nulls identical	Similar to PostgreSQL
	Set	**Order**
PostgreSQL	Primary keys	Unordered
SciDB	Dimensions	Dimensions Implicit
Neo4J	Specify Unique	Unordered
MongoDB	Uniqueness by index	Unordered
	Links	**Nesting**
PostgreSQL	Foreign keys	Arrays
SciDB	None	None
Neo4J	Nodes as links	Collections/array as prop.
MongoDB	Saved _id or DBRef	Used often

TABLE I: Examples Of Semantic Properties

the semantic properties mentioned above result in very different query formulation for similar tasks. A few examples are listed in Table II.

It is therefore imperative that we precisely identify the sets of equivalent operators and queries across different engines if the polystore system is to faithfully execute the user's queries. In this work, as part of ISTC's Big Data Analytics Working Group (BigDAWG), we inherit the BigDAWG polystore architecture and contribute to building a smart polystore optimizer through formulating the problem and presenting the terminologies useful for exploring semantic equivalent among operators from different back-end engines.

IV. BIGDAWG ISLANDS

An absolute equivalence between semantics of two database engines only exist when the two engines are identical; such equivalences would not be of much use for a polystore optimizer. On the other hand, equivalences of semantics within a limited context are possible. To address these kinds of equivalences, we introduce the notion of an *island*. An island in a BigDAWG polystore deployment specifies qualities that collectively describe a query language and its assumed data model. These qualities may include primitive types, data structures, operators implementations, handling of missing values, and so on. Islands are user-facing constructs and there are two types of them: degenerate and virtual islands. A degenerate island (DI) is an abstraction that captures semantics of a specific language and its underlying data model (e.g., PostgreSQL island, SciDB island). A virtual island (VI) captures a set of semantics that are analogous to those of the DI, but these islands are not grounded by any governing back end. A VI typically supports semantics that are *commonly available* in a class of database systems, such as relational database systems or graph database systems. It uses a more restricted variation of the most popular language adopted by its member engines as its island local language. Further, a polystore system may even offer purely virtual (user-defined) islands that enforce custom

	Filter
PostgreSQL	SELECT widget, type FROM inventory WHERE type = "PDA";
SciDB	filter(inventory, type = "PDA");
Neo4J	MATCH (n:Inventory) WHERE inventory.type = "PDA" RETURN n.widget, n.type;
MongoDB	db.inventory.find({type: "PDA" } , {_id: 0, widget: 1, type: 1})
	Equi-Join
PostgreSQL	SELECT widget, cost FROM inventory JOIN cost_april ON inventory.w_id = cost_april.w_id;
SciDB	cross_join(inventory, cost_april, inventory.w_id, cost_april.w_id);
Neo4J	MATCH (w_id)–[i:inv]–>(w_id)–[c:c_aprl]–>(w_id) RETURN widget, cost;
MongoDB	Only has built-in support for LEFT OUTER JOIN Returns a nested object db.inventory.aggregate([{$lookup: {from: "cost_april", localField: "w_id", foreignField: "w_id", as: "inventory_price"}}, {$match: {"inventory_price": {$exists: true, $not: {$size: 0}, $elemMatch:{"w_id":{$ne: null}}}}}]);
	Matrix Multiply
PostgreSQL	SELECT sum(m1.value*m2.value) FROM m1 JOIN m2 ON m1.row = m2.col GROUP BY m1.col, m2.row;
SciDB	spegmm(m1, m2);
Neo4J	MATCH (a:m)–[r1]–>(b:n)–[r2]–>(c:p) RETURN a.nid, c.nid, sum(r1.val * r2.val) ORDER BY a.nid, c.nid;
MongoDB	db.matrix2.aggregate([{$match : { $or : [{name: 'a'}, {name : 'b'}]}}, {$unwind : '$values'}, {$project : {cell : {$cond : {if: {$eq: ["$name", "a"] }, then: {x : '$row', y : '$values.col', valx : '$values.value'}, else: {x: '$values.col', y: '$row', valy :'$values.value'}}}}}, {$group : {_id : {x : '$cell.x', y : '$cell.y'} , valx : {$sum: '$cell.valx'}, valy : {$sum: '$cell.valy'}}}, {$project : {val : {$multiply : ['$valx', '$valy']}}}, {$sort : {'_id.x' : 1, '_id.y' : 1}}, {$group : {_id : '$_id.x', row : {$first : '$_id.x'}, values : {$push : {col : '$_id.y', value : '$val'}}}}])

TABLE II: Examples Of Equivalent Operators

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

language bindings. Since VIs are not committed to preexisting query languages, they are equipped with shims that translate query written in their local language into executable queries of each engine resident on the VI.

When a user issues a polystore query, it may refer to more than one islands. If it does, we denote a sub-query that completely occurs in a single island an *utterance*. To delimit nesting utterances, BigDAWG requires its user to declare the language encoding the utterance through a *scope* specification, and BigDAWG allows its users to use *cast* specifications to optionally specify how intermediary results are transformed from the data model of one utterance to the next. Casts also become very useful when a user want to format and present the end result conforming to a certain data model.

V. Semantic Equivalences

Semantically equivalent queries from different islands are substitutable. Therefore by replacing an utterance into its equivalence in the semantics of its super-query, we create an opportunity for query optimization. In this light, we dedicate this section to discuss semantic equivalences of queries.

In an ideal world, the universe of our polystore semantics could be represented by the drawings in Figure 3: The shaded

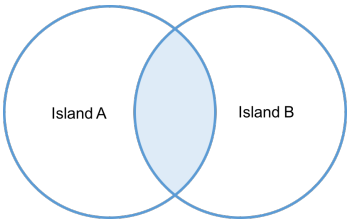


Fig. 3: Ideal Distribution of Semantic

area would be well-known semantics that are available in every DI and we can add as many systems as we want to it, provided that they offer the semantics implied by the shaded region. This area thus describes an ideal class of semantic equivalences to use in query optimization. In reality, however, we are more likely to encounter a polystore system like the one shown in Figure 4. We will probably find section ABCDE

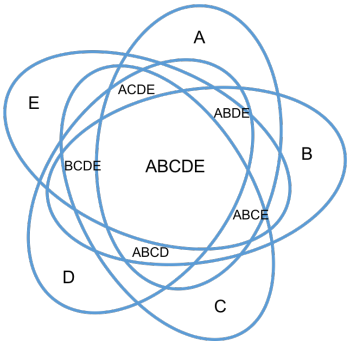


Fig. 4: Realistic Distribution of Semantics

to cover the basics, e.g. filters and simple aggregates, and section ABCD to cover less common operators, such as path finding between fixed ends and sparse matrix multiplication. Taking this abstraction of classes of shared semantics a step further, we generalize and transform the Venn Diagram into a lattice structure, as shown in Figure 5: Each node on the bottom layer of the lattice represents a DI, and the ones in the second layer from the bottom represent the VIs. Further up the lattice, each node represents a set of semantics shared by the nodes that it connects to from below. In other words, the nodes in the upper lattice denote equivalent semantics across islands. Two vertically neighboring nodes on the lattice differ by their expressiveness: the higher one contains fewer but more widely supported operators; whereas the lower ones support more expressive semantics across fewer DIs.

Sometimes a sub-query composed of operators from lower level nodes may be rewritten in more basic operators from higher level nodes. This type of rewrite creates opportunities to integrate a sub-query with its super-query, which give rise to new options for optimization. On the other hand, a rewrite that make use of less commonly supported operators may provide performance benefits. A polystore optimizer should be able to identify the rewrites that offer the most performance advantages for executing the entire query across islands.

Naturally, the polystore system should not burden end users with details of the lattice's layout. Rather, an end user should be able to write polystore queries in the island languages of its choice and leave it to the polystore optimizer to optimally move sub-queries between the levels of the lattice. If the user offers an utterance with a scope, the optimizer will compile the utterance into a direct acyclic graph (DAG) of operators and trace through the lattice to determine the placement of the utterance. If the utterance arrives without an explicit scope, the optimizer would work through the lattice top-down, identifying the VIs that are capable of interpreting the statement and alerting the user in case of semantic ambiguity.

When conducting query optimization of nesting semantics from different islands, the optimizer may break the DAG of a query into multiple pieces, each composed of operators from a common node in the lattice. This allows the optimizer to rearrange and consolidate sub-queries into executable queries for single DIs. Finding the right levels on this lattice of semantic offerings to house each component of the original DAG remains an interesting problem open for solutions.

VI. Rules For Semantic Equivalences

Each node on the lattice will be a host for a collection of rules depicting the context and the semantic equivalences occur in the context. We will denote the collection a *dictionary*, and the individual rules *equivalence rules*. Take sparse matrix multiplication for an example: in the DI of SciDB, its AFL language expresses the query as the following[1]:

```
multiply(a, b);
```

[1]The AFL syntax used in SciDB 14.12 for sparse matrix multiplication is "spgemm(a, b)"; here, we use a legacy notion "multiply(a, b)" for its better readability.

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

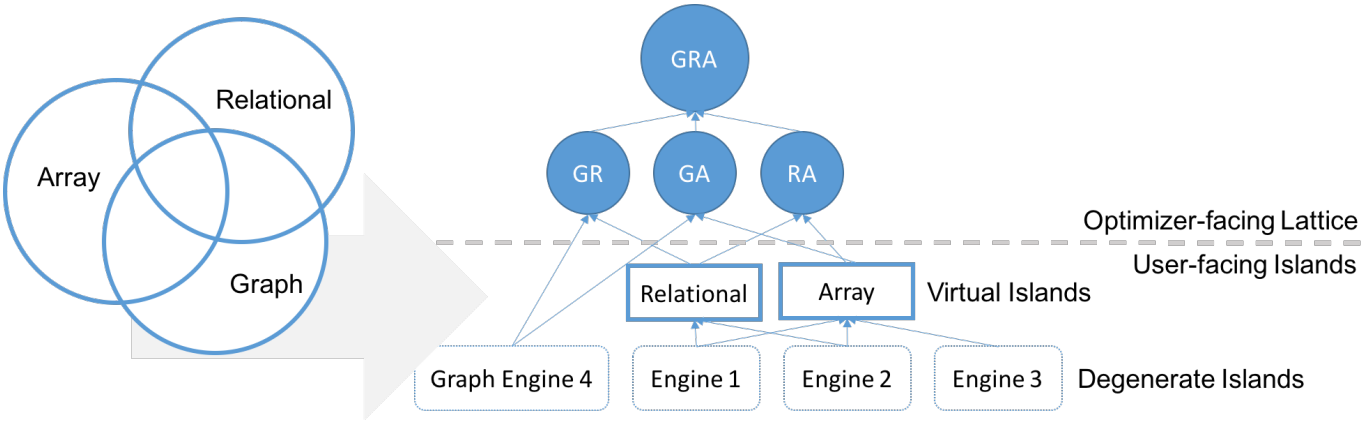
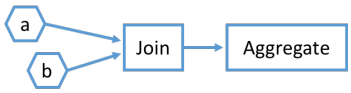


Fig. 5: A Semantic Lattice

In the DI of PostgreSQL using SQL, the same query is expressed as:

```
SELECT a.row_num, b.col_num,
       SUM(a.value*b.value)
FROM a, b
WHERE a.col_num = b.row_num
GROUP BY a.row_num, b.col_num;
```

Rewriting the SQL query in the form of a DAG, it becomes:



We ran the two queries on their respective back-end databases with the same data input, and we compared the results to learn about the similarity between the queries. After running the two queries over different sets of engineered inputs, we concluded that the two queries return the same set of values, but orders of the result entries might differ. Therefore, we construct a dictionary entry as follows:

```
{AFL:multiply(a, b);
 SQL:aggregate(join(a, b));
 "All values for (int64, integer)"}
```

The construction of dictionary entries should be automated; yet the infinite search space in comparison to the handful of truly interesting queries makes it a prohibitive problem. To make the problem manageable, a semi-automatic test-based approach should be adopted. Specifically, we will ask human curators to provide queries to be compared, and a testing program to verify their equivalence properties as well as to suggest variations of the original queries to explore subsequently. A refinement of this approach will make an interesting discussion for another time.

VII. Semantic Containment In Place Of Equivalence

An equivalence between two operators from different data models is described by producing the same output given equal input data. A large dictionary consisting equivalence rules will allow us to build a power polystore optimizer. However, we would like to take a step further in the discussion of equivalences and expand the space of possible polystore optimizations. To do so, we introduce the concept of *containment*–in terms of operators, an operator A is contained by an operator B if any input of A migrated to the island containing B allows B to produce the same results as those produced by A. In other words, the concept of containment emulates equivalence, but without the symmetry.

We differentiate among different types of containment using the characteristics of translated operators and queries. Said characteristics include whether the equivalence ensures consistency of orders among the output entries, whether information regarding the presence of missing values is recoverable after translation, and so forth. Distinguishing among different types of containment paves way to greater choices of translation in face of permissive contexts. We will discuss three types of containment below.

A. Orders Of Result Entries

An equivalence between two operators implies that they produce equal output and in identical order. However, queries and, more commonly, sub-queries in many cases do not require the result entries to be ordered. In this circumstance, two queries whose results differ only in orders should still be considered substitutes. When rewriting an AFL query in SQL, the translated sub-queries need not be specifically ordered, whereas the eventual output of SQL query should be sorted along the dimension columns used in AFL.

B. Expressiveness Of Semantics

Some operators or functions of an engine are implemented to be more expressive than their counterparts in other engines. For instance, fuzzy text search in most SQL systems is carried out in the form of the $LIKE$ expression. Such search mechanism only allows the user to specify what is present and how are they ordered. NoSQL systems frequently enlisted support of Regular Expressions (Regex) to facilitate the tasks of text search. This greatly expanded on the types of patterns

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

that could be matched from what LIKE expression provides. In this case, a LIKE expression could be accurately rewritten in Regex, whereas the reverse is not true.

C. Backward Compatibility For Primitive Types

Supporting legacy systems with older primitive types could be problematic. For instance, consider the interaction between a modern array store A and a legacy relational database B. A supports all types of integers, from 16-bit short to 64-bit long, and B only supports integers with maximum length of 32-bits. A non-nullable 32-bit integer column in B could be safely migrated to a non-nullable 32-bit or 64-bit integer attribute in A–even syntax for query the original 32-bit integer column could be directly applied to the new 32-bit or 64-bit integer attribute. Mapping an originally 64-bit integer attribute from A to B, however, should not be permitted because values of 32-bit integer is a strict subset of values of a 64-bit integer attribute. We term it that the 32-bit integer column in B contained by the 64-bit integer attribute in B.

VIII. Related Work

Researches on federation of relational and object-oriented database systems have yield significant results since 1980. Many works were devoted toward reconciling data scattered among relational database instances and the likes of object-oriented database instances [4] [5] [6] [7] [8] [9]. In contrast to the BigDAWG approach, which does not try to unify all semantics under a common language or data model, optimization in said federation systems commonly involves minimizing cost function subject to a query written in a standardized language and data model.

For example, TISMMIS [4] assumes a standard query language and data model and converts objects from all underlying data sources into the standard format during execution, and it relies on a mediator component to reconcile results returned from its multiple translators.

While some of the aforementioned federation systems conduct optimizations by considering details of all sub-queries within an input query, the approach of Garlic [5] is similar to that of BigDAWG in the sense that it advocates exploiting existing query optimization capabilities of the underlying engines. However, Garlic differs from BigDAWG since that it is mostly agnostic to the capabilities of a member database until it communicates with the engine after the sub-query is executed. Besides, though claiming to support non-relational engines, Garlic adopts the SQL syntax and data structures in the form of parsed SQL queries as the its sole intermediary media.

In addition to the federated database systems, we also reviewed the formulation and usages of KOLA [12], an algebra that serves as a basis for formulating query rewrite rules, and the COKO language, [11] which specifies COKO transformations that package KOLA-based rules with firing algorithms to rewrite complex queries. Though that both KOLA rules and COKO transformations are conceived in light of relational and object-oriented data models, their approach could be modified to conduct equivalence explorations in the context of polystore for languages whose semantics are known.

IX. Conclusion

The emergence of the polystore system brings new challenges and opportunities in terms of query optimization. To address these challenges and opportunities, we formulated the optimization problem in the context of the BigDAWG polystore specification, demonstrated the semantic lattice, a framework for organizing semantic equivalent operators and queries to construct a decision space for polystore query optimizations, and presented semantic containment, asymmetric relations used in place of symmetric equivalence to expand the decision space. Through these contributions, we developed a foundation for creating an effective polystore query optimizer.

Acknowledgment

The authors would like to thank Intel Science and Technology Center for Big Data for their support of this work. The authors would also like to thank colleague Dylan Hutchinson from University of Washington for providing feedback to an earlier draft of this work.

References

[1] J. Duggan, O. Papaemmanouil, L. Battle, and M. Stonebraker, *Skew-Aware Join Optimization for Array Databases*. In Proceedings of SIGMOD, 2015.

[2] V. Gadepally, J. Duggan, A. Elmore, J. Kepner, S. Madden, T. Mattson and M. Stonebraker, *The BigDAWG Architecture*. New England Database Summit, 2016.

[3] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, et al., *A Demonstration of the BigDawg Polystore System*. Proceedings of the VLDB Endowment, 2015.

[4] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman and J. Widom, *The TSIMMIS project: Integration of heterogenous information sources*, 1994.

[5] V. Josifovski, P. Schwarz, L. Haas and E. Lin, *Garlic: a new flavor of federated query processing for DB2*. In Proceedings of the 2002 ACM SIGMOD international conference on Management of data, pp. 524-532, June 2002.

[6] R. Ahmed, P. DeSmedt, W. Du, W. Kent, M. A. Ketabchi, W. A. Litwin and M. C. Shan,. *The Pegasus heterogeneous multidatabase system*. Computer, 24(12), pp. 19-27, 1991.

[7] M. Templeton, D. Brill, S. K. Dao, E. Lund, P. Ward, A. L. Chen and R. MacGregor. *Mermaid?A front-end to distributed heterogeneous databases*. Proceedings of the IEEE, 75(5), pp. 695-708, 1987.

[8] G. Gardarin, F. Sha and Z. H. Tang, *Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System*. In VLDB (Vol. 96), pp. 3-6, September 1996.

[9] A. P. Sheth and J. A. Larson, *Federated database systems for managing distributed, heterogeneous, and autonomous databases*. ACM Computing Surveys (CSUR), 22(3), pp. 183-236, 1990.

[10] I. Robinson, J. Webber and E. Eifrem, *Graph Databases: New Opportunities for Connected Data*, "O'Reilly Media, Inc.", pp. 153, 2015.

[11] M. Cherniack and S. Zdonik, *Changing the rules: Transformations for rule-based optimizers*. In ACM SIGMOD Record (Vol. 27, No. 2), pp. 61-72, ACM, June 1998.

[12] M. Cherniack and S. B. Zdonik, *Rule languages and internal algebras for rule-based optimizers*. In ACM SIGMOD Record (Vol. 25, No. 2), pp. 401-412, ACM, June 1996.

978-1-5090-3525-0/16/$31.00 ©2016 IEEE