Enabling Query Processing across Heterogeneous Data Models: A Survey

Ran Tan, Rada Chirkova
Department of Computer Science
North Carolina State University
Raleigh, North Carolina
Email: rtan2@ncsu.edu, rychirko@ncsu.edu

Vijay Gadepally
Lincoln Laboratory
Massachusetts Institute of Technology
Lexington, Massachusetts
Email: vijayg@ll.mit.edu

Timothy G. Mattson
Intel Corporation
Portland, Oregon
Email: timothy.g.mattson@intel.com

***Abstract*—Modern applications often need to manage and analyze widely diverse datasets that span multiple data models [1], [2], [3], [4], [5]. Warehousing the data through Extract-Transform-Load (ETL) processes can be expensive in such scenarios. Transforming disparate data into a single data model may degrade performance. Further, curating diverse datasets and maintaining the pipeline can prove to be labor intensive. As a result, an emerging trend is to shift the focus to federating specialized data stores and enabling query processing across heterogeneous data models [6]. This shift can bring many advantages: First, systems can natively leverage multiple data models, which can translate to maximizing the semantic expressiveness of underlying interfaces and leveraging the internal processing capabilities of component data stores. Second, federated architectures support query-specific data integration with just-in-time transformation and migration, which has the potential to significantly reduce the operational complexity and overhead. Projects that focus on developing systems in this research area stem from various backgrounds and address diverse concerns, which could make it difficult to form a consistent view of the work in this area. In this survey, we introduce a taxonomy for describing the state of the art and propose a systematic evaluation framework conducive to understanding of query-processing characteristics in the relevant systems. We use the framework to assess four representative implementations: BigDAWG [7], [8], CloudMdsQL [9], [10], Myria [11], [12], and Apache Drill [13].**

***Keywords*-Cross-model query processing; Query-specific data integration; Taxonomy; Evaluation framework**

I. Introduction

Modern applications often need to manage and analyze widely diverse datasets that span multiple data models. In medical informatics [1], [2], health professionals serve patients admitted to intensive-care units using data expressed as structured demographics, semi-structured laboratory and microbiology test results, discharge summaries, radiology, cardiology reports in text formats, and vital signs and other data in time-series format. In oceanographic metagenomics [3], biologists detect relationships between cyanobacteria communities and environmental parameters via integrating genome sequences, structured sensor and sample metadata, cruise reports in text formats, and streaming data generated by flow-cytometer systems. In intelligent transportation management [4], administrators analyze open traffic data presented in RDF (Resource Description Framework), city events expressed as JSON (JavaScript Object Notation) documents, social-media data recorded via key-value pairs, and weather feeds stored in relational tuples to predict traffic flows. Finally, in data journalism [5], journalists work with Tweet texts, relational databases provided by governments and institutions, and RDF-formatted Linked Open Data to support content management for writing political articles.

In these and other scenarios, warehousing the data using Extract-Transform-Load (ETL) processes can be very expensive. First, transforming disparate data into a single chosen data model may degrade performance. Indeed, there appears to be no "one size fits all" solution for all markets [14], [15], as specialized models and architectures enjoy overwhelming advantages in data warehousing, text searching, stream processing, and scientific databases. Second, curating diverse datasets and maintaining the pipeline could turn out to be labor intensive [16]. One major reason is that rules and functions in ETL scripts do not adapt to changes in data and analytical requirements, and changes in application logic often result in the modification of ETL scripts.

For these and other reasons, a number of projects are shifting the focus to federating specialized data stores and enabling query processing across heterogeneous data models [6]. This shift can bring many advantages. First, the systems can build natively on multiple data models, which can translate to maximizing the semantic expressiveness of underlying interfaces and to leveraging the internal processing capabilities of each data store. Typical tasks can be expressed natively in a variety of algebras, such as relational, linear, and graph algebra, and be executed economically on a variety of specialized data stores optimized for different workloads. Second, federated architectures support query-specific data integration with just-in-time transformation and migration, which has the potential to significantly reduce the operational complexity and overhead. Data transformations across data models and data migration between data stores can be explicitly expressed via queries and automatically handled by the system, bridging the gap between data preparation and data analysis.

Projects that focus on developing systems in this research area stem from various backgrounds and address diverse concerns, which can make it difficult to form a

consistent view of the work in the area. Some of the projects concentrate on the issues of semantic mapping and record linkage; some define operators over multiple data models and focus on multi-model query planning and optimization; others emphasize data-flow optimization and multi-platform scheduling. Such diverse perspectives and viewpoints add to the complexity of understanding the field, and might even cause unnecessary miscommunication between research groups. Therefore, it would be beneficial to have a taxonomy of the field that would contribute to clear definitions of the key terms. We could then build on the taxonomy by specifying an evaluation framework focused on the query-processing characteristics of each design.

This paper makes the following specific contributions:

- We introduce a taxonomy that categorizes state-of-the-art solutions in federated database systems, polyglot systems, multistore systems, and polystore systems;
- To understand the query-processing characteristics of each design, we propose an evaluation framework incorporating the axes of heterogeneity, autonomy, transparency, flexibility, and optimality;
- We survey current efforts in the integration of disparate data, and provide an analysis of query-processing characteristics for four representative systems; and, finally,
- We discuss major challenges and future directions in query processing across heterogeneous data models.

II. Taxonomy

Systems federating specialized data stores and enabling query processing across heterogeneous data models can be characterized by the data stores and query interfaces that they support. We introduce a taxonomy that builds on this observation and groups state-of-the-art solutions into four categories, defined as follows:

- A *federated database system* comprises a collection of homogeneous data stores and features a single standard query interface.
- A *polyglot system* hosts data using a collection of homogeneous data stores and exposes multiple query interfaces to the users.
- A *multistore system* is able to manage data across heterogeneous data stores, while supporting a single query interface.
- A *polystore system* enables query processing across heterogeneous data stores and supports multiple query interfaces.

A. Federated Database Systems

Federated database systems have been extensively studied by the data-research community since the 1980s. Such systems often feature a mediator-wrapper architecture, and employ schema-mapping and entity-merging techniques for integration of relational data. A representative system called Multibase [17] defines a global schema, a mapping language, and a mechanized local-to-host translator. Users pose queries against the global schema, which are then mapped to the local schemata and an integration schema; the translator translates the decomposed queries into native queries. Over the years, semantic heterogeneity has remained a challenge for federated database systems, as well as for other systems employing the federated architecture.

B. Polyglot Systems

Polyglot systems are designed to take advantage of the semantic expressiveness of multiple interfaces. It is largely driven by the need to manage complex data flows in distributed file systems, where data pipes need to be expressed in both declarative queries and procedural algorithms. As an example, Spark SQL [18] provides a DataFrame API, which allows users to access data in the relational and procedural modes. Spark SQL supports SQL statements, and passes operations to a relational optimizer named Catalyst to achieve high performance. This integration of relational and procedural APIs makes Spark SQL an excellent tool for complex analytics. An advantage of polyglot systems is that different query interfaces might provide compensating expressiveness and could greatly simplify query formulation.

C. Multistore Systems

Multistore systems have been proposed to address the challenge of querying large-scale collections of heterogeneous data stores. Such systems are designed to address various concerns, and can thus be categorized into several groups. One of the directions, with systems including HadoopDB [19], Polybase [20], and JEN [21], focuses on integrating distributed file systems with relational database systems. HadoopDB extends HIVE [22] to push-down operations into multiple single-node DBMS instances connected through a MapReduce layer. Polybase comprises a data warehouse and HDFS with a common parser, and translates SQL operators into MapReduce jobs for data in HDFS. JEN leverages a sophisticated execution engine to do most data processing on the HDFS side. Another group of solutions integrates a variety of NoSQL databases with relational databases, as exemplified by BigIntegator [23], Forward [24], and D4M [25]. In this collection, BigIntegrator allows querying data stored both in relational databases and in various cloud-based data stores, by using a subset of SQL named GQL. Forward employs a JSON-based data model, while D4M employs the associative-array data model to enable querying NoSQL data stores. Systems focusing on dealing with data placement in the multistore setting, including ESTOCADA [4], Odyssey [26], and MISO [27], use materialized views to optimize data placement across data stores for query performance. Systems that adopt the semantic approach and use ontologies to mediate relational and non-relational data sources, including TATOOINE [5] and OPTIQUE [28], provide a unified querying layer over

multiple data stores, by adopting ontologies and applying schema-mapping and entity-resolution techniques.

D. Polystore Systems

A polystore system combines the advantages of the polyglot and multistore sytems: Users can choose from a variety of query interfaces to seamlessly query data residing in multiple data stores. Among polystore systems, BigDAWG [7], CloudMdsQL [10], Myria [11], and Apache Drill [13] focus primarily on query answering, while QoX [29], Musketeer [30], and Rheem [31] concentrate on multi-platform data-flow scheduling and analytics. AWESOME [32] takes a different perspective, by focusing on challenges in data ingestion and derivation with heterogeneous data stores.

The above taxonomy can serve as a high-level tool for identifying and classifying systems based on the descriptive feature of heterogeneity in data stores and query interfaces. In practice, different systems are designed with different emphases on the choice of computational models and languages, execution engines, and data-store technologies. To provide a better understanding of the query-processing characteristics of each individual system, we have developed a finer-grained evaluation framework, which we present in the next section.

III. The Evaluation Framework

While several attempts have been made to address data-store federation and query processing across heterogeneous data models, to the best of our knowledge, there is no consensus on an evaluation framework that would provide context and serve as a reference. We propose to fill the gap with a framework that could help researchers navigate the design choices in information-integration systems. The proposed framework, inspired by the work [33] on federated database-management systems, consists of five dimensions – heterogeneity, autonomy, transparency, flexibility, and optimality, as detailed below.

A. Heterogeneity

In data-integration systems, the design intent is threefold: Seamless access to and management of the data in the underlying data stores, leverage of the internal processing capabilities of the component processing engines, and minimal loss of expressiveness of the underlying query interfaces. Accordingly, heterogeneity can be measured in three directions:

1) Data-Store Heterogeneity: Data stores with different modeling techniques and physical architectures are good fits for different workloads: In data-warehousing workloads, well-tuned column stores hold an order-of-magnitude advantage compared to traditional row stores, while in text searching and retrieval, key-value stores can perform many times better than RDBMS.

2) Processing-Engine Heterogeneity: Processing-engine heterogeneity reflects the processing capabilities of integration systems. Processing engines modeled around arrays, graphs, and dictionaries often complement relational database engines in the specialized functionalities. A processing engine modeled on relations, arrays, and graphs can handle complex linear-algebra operations and pattern-matching tasks better than traditional relational database engines.

3) Query-Interface Heterogeneity: Adoption of different data models in query engines indicates applicability of various formal algebras, each promising different expressive capabilities, with each interface providing its own semantic context. A system providing both the relational and array interfaces would enable the users to express simple linear-algebra operations on relational data. Query-interface heterogeneity reflects the semantic expressiveness of such integration systems.

B. Autonomy

In practice, data sharing often takes place under regulations and constraints. In many cases, disparate databases are integrated in the form of a conceptualized federation, where a certain level of autonomy applies to each component database-management system.

1) Association Autonomy: A database-management system may decide when to associate and disassociate itself from the federation. The integration system should be able to adjust to the dynamic environment, with individual databases joining and leaving the federation over time.

2) Execution Autonomy: Component database-management systems may continue supporting native applications while participating in a federation. Thus, native queries can be executed without interference from the integration system. When native and integration queries compete for resources, the component database-management systems decide on the relative execution priorities.

3) Evolution Autonomy: Each component database may continue to evolve in its own business models and schemata. The integration system is required to adapt to the changes and to stay operational during the evolution.

C. Transparency

Systems can achieve query-specific integration with just-in-time transformation and migration, by providing a consistent programming model and user-friendly query interface. Integration across heterogeneous data models would be a labor-intensive activity in case one has to work around the details of storage and data layout, instead of having transparent access to data.

1) Location Transparency: Distributed and parallel databases often partition and replicate data sets among multiple nodes. A data set might also span multiple storage engines with different data models. Providing transparent

access to distributed data sets is a nontrivial problem. Proper abstractions, such as treating data sets as variables and referencing them with variable names, help hide location details.

2) Transformation Transparency: Data transformations between data models and migration between heterogeneous database engines are unavoidable in integration. During a transformation, a transparent system would actively infer and map data types, and also adjust data structures, by, e.g., flattening a nested data structure to fit a table. This way, users can focus on logical-level transformations without having to worry about the details of data types and structures.

D. Flexibility

The popularity of Hadoop-style ecosystems largely relies on their flexibility in handling various analytical tasks [34]. Instead of providing a single data model and programming model, Hadoop develops a stack of systems that helps manage data in arbitrary formats and supports flexible workflows and user-defined functions.

1) Schema Flexibility: Traditional database-management systems require pre-processing and cleansing of raw data before ingestion. This process can be automated with user-defined schemata and dynamic schema discovery. Minor changes in formats and semantics can be automatically detected and transformed.

2) Interface Flexibility: Query interfaces are often designed around fixed algebras. Such designs naturally limit the expressiveness of the query interface. Allowing user-defined functions and extensibility via embedded queries and constrained imperative programmability results in more flexibility in expressing complex tasks.

3) Architectural Flexibility: Modularized architectures with extensibility to external query interfaces, query optimizers, back-end engines, and other functionalities would make the systems customizable to a wide range of scenarios.

E. Optimality

Query processing across heterogeneous data models offers rich opportunities for optimization. The participating database engines each suit best different workloads. Optimizing data placement and generating federated query plans could result in significant performance improvements.

1) Federated Plan Optimization: Federated query plans can be executed by pushing subqueries down to specialized data stores. With cross-model query equivalence and containment, the optimizer could transform data and migrate them across data stores to achieve better performance. When the complexity of the analytical task outweighs the transfer cost, notable performance improvements are possible.

2) Data-Placement Optimization: Proper placement of data across a collection of heterogeneous data stores could take advantage of each processing engine while saving on transfer time. Various rule-based and cost-based methods can be applied to data-placement optimization.

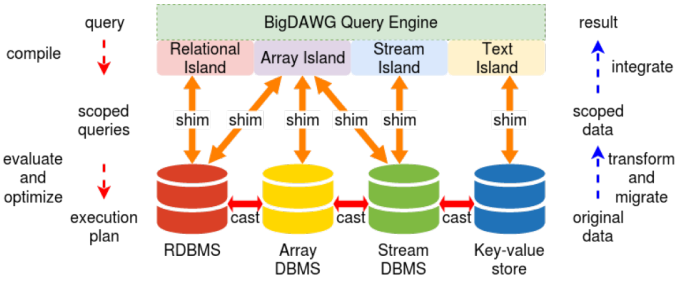

Figure 1. The BigDAWG architecture, adapted from [7].

```
RELATIONAL(SELECT *
    FROM R, CAST(A, relation)
    WHERE R.v = A.v);
```

Figure 2. BigDAWG example queries [7].

IV. Implementations

Due to the page limit, we focus here only on those polystore systems whose design and development emphasize addressing query-processing and query-answering challenges, toward fulfilling the promise of enabling effective query processing across heterogeneous data models. The selection of system is representative to the best of our ability.

A. BigDAWG

1) Overview: BigDAWG [7], [8] introduces information islands as an abstraction layer. Each island specifies a data model and its logical structure, a query language or algebra for that data model, and one or more back-end engines for data storage and query execution.

Users pose queries in the scope of a specified information island. The query will then be decomposed and mapped into native subqueries that can be executed by database engines connected to the island. When an analytical task cannot be easily expressed in a single island's semantics, the users need to specify multiple island languages with **SCOPE** operators. Often, a **CAST** operator is used to change the semantic context in a cross-island query. Heterogeneous database engines are supported by connecting to one or more information island via wrappers called shims.

2) BigDAWG Query Processing: The BigDAWG middleware consists of four modules [8]: the query-planning module (planner/optimizer) [35], the performance-monitoring module (monitor) [36], the data-migration module (migrator) [37], and the query-execution module (executor) [38]. These four modules collaborate to support cross-data-model queries. When BigDAWG receives an incoming query, the planner parses the query to generate an ordered list of viable query-plan trees, with possible engines for each collection of data objects. The monitor then applies existing performance information to each query-plan tree, to determine the best engine for each query-execution tree. The query-plan tree

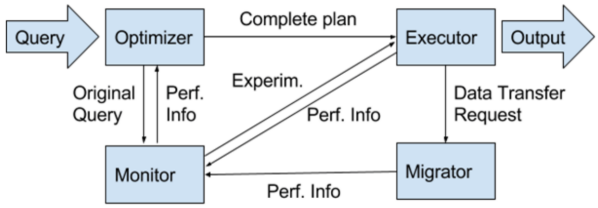

Figure 3. BigDAWG query processing [8].

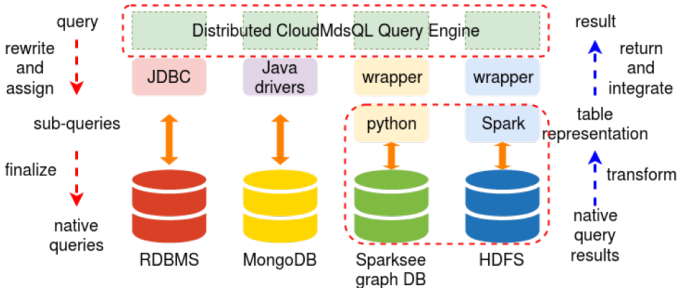

Figure 4. The CloudMdsQL architecture, adopted from [9].

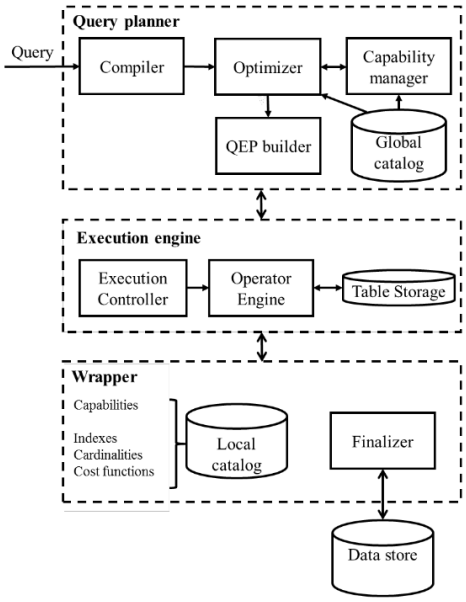

Figure 6. CloudMdsQL query processing [9].

is then passed to the executor for best join policies. The migrator moves data across database engines as needed.

B. CloudMdsQL

1) Overview: CloudMdsQL is a scalable SQL query engine with extended capabilities for seamless querying of heterogeneous non-relational database engines. The query language is SQL based, with embedded functional sub-queries written in the native query languages of the underlying database engines. The query engine is fully distributed and collocated at the node for each database engine in a cloud environment; this enables direct communication across compute nodes in the exchange of query plans and data. CloudMdsQL also employs transparent wrappers on the query interface of each database engine, as well as a table-based common data model for storing intermediate results and exchanging data.

2) CloudMdsQL Query Processing: Each query is first processed by the compiler to generate an Abstract Syntax Tree (AST), which corresponds to the syntax clauses in the query. CloudMdsQL then identifies a collection of subtrees

```
T1(x int, y int)@rdb =
    (SELECT x, y FROM A)
T2(x int, z array)@mongo = {*
    db.B.find({$lt: {x, 10}}, {x:1, z:1})
*}
SELECT T1.x, T2.z
FROM T1, T2
WHERE T1.x = T2.x AND T1.y <= 3
```

Figure 5. CloudMdsQL example queries [9].

from AST, each associated with a data store and labeled by a named table. Subtrees for non-SQL subqueries are passed to the corresponding wrappers and translated into the native language of each data store. The native queries are then pushed down to the respective query engines for execution. The rest of AST is handled by a SQL query engine.

After the lexical and syntactic analyses, the query planner resolves the names of tables and columns with signatures, checks for data-type compatibility between the operands, and infers the data types to be returned. It also attempts to rewrite query plans and to perform cross-reference analysis, to make sure there are no cycles in the dependency graph.

The optimizer gets cost functions from the global catalog and implements a simple exhaustive-search strategy to select the best query plan. The capability manager validates each rewritten sub-query against its data-store capability specification in the global catalog. Then the query-execution plan (QEP) builder generates query-execution plans from the best query plans, and serializes the plans to JSON. The query-execution controller parses QEP, identifies the sub-plans associated with named table expressions, and executes **CREATE FUNCTION** statements to invoke the wrapper. The operator engine caches into the table storage those intermediate results that are needed more than once. A specific query plan is produced in byte code. Wrappers store local catalog information and provide it to the query planner periodically or on demand. A finalizer for the database engine translates the subquery plans into native queries that can be executed by the engine.

C. Myria

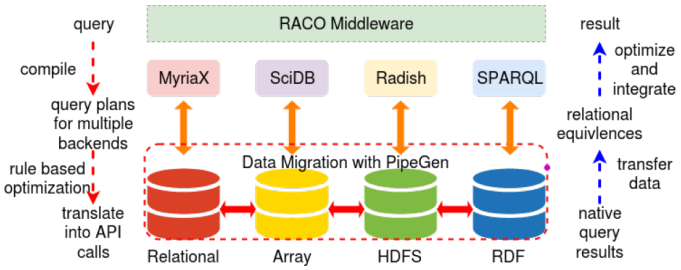

Figure 7. The Myria architecture, adopted from [11].

```
T1 = scan(A);
T2 = scan(B);
def foo(a, b): a - b;
joined = SELECT *
         FROM T1, T2
         WHERE foo(T1.a, T2.b) = 0;
store(joined, C);
```

Figure 8. Myria example queries, adopted from [11].

1) Overview: At its core, Myria is designed as a federated data-analytics system, with an imperative-declarative hybrid language, MyriaL, to facilitate expressing complex data analytics. Each declarative MyriaL statement can be wrapped with imperative constructs, such as variable assignments and iterations. To cope with large collections of domain-specific scripts written in Python and various scientific computation workflows, Myria supports user-defined functions (UDF) and aggregates (UDA) via an exposed Python API. The built-in query-execution engine, MyriaX, adopts the traditional shared-nothing parallel database-system architecture, which inherits the benefits of the associated optimization techniques. The engine is enhanced with crucial features in modern big-data analytics, including iterative processing, elasticity to horizontal scalein and scaleout to many servers, data ingestion from HDFS and cloud storage, and deployment on top of resource managers such as YARN [39].

The Relational Algebra Compiler (RACO) serves as Myria's query optimizer and federated query executor; it uses relational algebra extended with imperative constructs capturing the semantics of array, graph, and key-value data models. RACO accepts queries in MyriaL and generates query plans for both MyriaX and the selected array, graph, and key-value engines, including Spark, SciDB, and PostgresQL. This allows Myria to generate federated query plans that take advantage of individual specialized database engines. It also employs PipeGen [40] to automatically enable efficient data migration between arbitrary storage engines, in support of query plans across engine boundaries.

2) Myria Query Processing: Queries to Myria can be expressed either in the MyriaL language or through the Python API. MyriaL is extended with imperative constructs, such as assignment statements and iterations, to support user-defined functions and user-defined aggregates. In addition, Python UDFs and UDAs can be registered and called in MyriaL to directly operate on the blob data type in the MyriaX query-execution engine. The integrated Python API enables query composition through successive invocations of functions on relational objects. All types of queries are parsed into the Myria algebra and then transformed into the specific API calls, operators, or query primitives supported by the selected database engines in RACO. This often involves defining semantic equivalences and adding translation rules. RACO then uses rule-based optimization to generate federated query plans that take advantage of the performance characteristics of the supported database engines. By default, the optimizer assigns each leaf of the plan to the platform on which the dataset resides, and then iterates from the bottom up, inserting the data-movement operator whenever the child nodes reside on different platforms. Optimized data transfers between arbitrary database engines are supported by PipeGen [40].

```
SELECT tbl1.id, tbl1.type
  FROM mongodb.doc AS tbl1
  JOIN dfs.`.../file.json` AS tbl2
  ON tbl1.id = tbl2.id;
```

Figure 9. Apache Drill example queries, adopted from [41].

D. Apache Drill

1) Overview: Apache Drill is a fully distributed, massively parallel query engine that supports low-latency interactive ad-hoc analysis at scale. It is designed for extensibility, with well-defined APIs for pluggable query languages, query planners, query optimizers, and storage plugins. The query engine accepts ANSI SQL and MongoDB QL, as well as user-defined functions and custom operators, and supports HBase, Hive, MongoDB, relational databases, and various file systems.

The success of Apache Drill relies on two factors: (i) the in-memory columnar hierarchical data representation, based on Parquet and JSON, that achieves both the efficiency of the columnar-stripped data model and the flexibility of the schema-free nested data model, and (ii) the fully distributed query engine, powered by Apache Calcite, that compiles and recompiles queries dynamically based on real-time data to maximize data locality and parallel processing, and enables on-the-fly schema discovery and in-situ data querying.

2) Apache Drill Query Processing: Apache Drill consists of a daemon service called Drillbit running on any or all nodes of a Hadoop cluster. Each Drillbit retains the full services and capabilities of Drill. The Zookeeper [42] managing the cluster serves as a broker between the client and Drillbits, as well as among Drillbits. When a user issues a query to Drill, the client first contacts Zookeeper for an available Drillbit, to which the query is then submitted. The

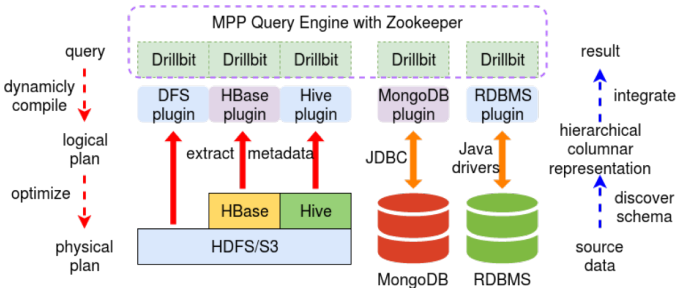

Figure 10. Apache Drill architecture, adopted from [13].

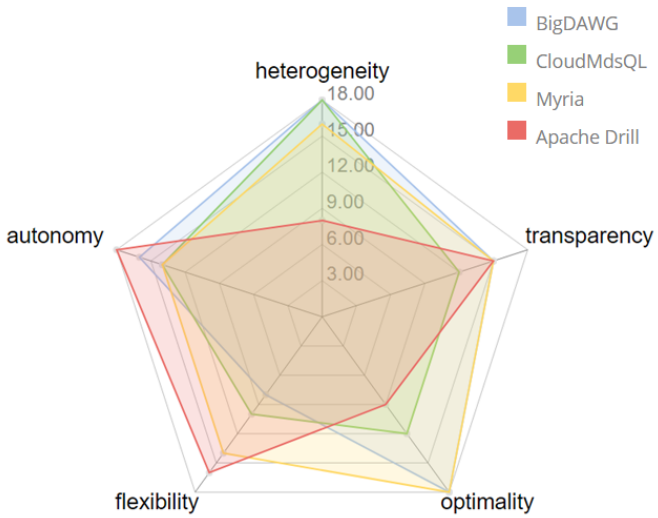

Figure 11. Visualizing the evaluation results; see Sec. IV-E for the details.

Drillbit that accepts the query becomes the Foreman that parses the query to generate a language-agnostic, computer-friendly, tree-structured logical plan that optimizes the abstract dataflow for data locality and parallel processing. The Drill optimizer then determines the best execution plan and translates the logical plan into a physical plan that describes the physical operations carried out by the execution engine. The physical plan is then rendered into a number of fragments organized into a multli-level execution tree. Each leaf execution engine processes the data and assembles them into an in-memory columnar hierarchy representation. The parent execution engines integrate the intermediate results to produce the answer. It is noteworthy that the storage plugin for Hive takes advantage of the metadata abstraction layer and operates on the files without invoking Hive's execution engine, to avoid unnecessary latency. Other storage plugins may take advantage of the query engine of the connected data-management system, to leverage the internal processing capabilities.

E. Summary

We have qualitatively evaluated each system against each aspect of the proposed framework; the results are summarized in Table I. To visualize the results, we have shaded each evaluation result in Table I by red, yellow, or green. Here, red stands for "little support," yellow stands for "partial support," and green stands for "excellent support." Further, we have rendered the query-processing characteristics of the four systems in a radar chart, to compare and contrast their features along the five dimensions, as shown in Figure 11. We assign a score of 1 to features with little support, 2 to partially supported features, and 3 to features with excellent support. The scores along each dimension are then combined and normalized to a scale from 0 to 18. Connecting the points on the radar chart results in a pentagon area that represents the query-processing characteristics of each system.

While not meant to be comprehensive, the proposed evaluation framework attempts to capture some of the crucial query-processing characteristics of integration system. Though the evaluation process can be termed subjective, the framework enables an intuitive glimpse into the query-processing characteristics that could otherwise be hard to perceive. For example, the framework could assist users in choosing proper data-management tools for their applications, via mapping their application needs onto a requirement pentagon, to enable selection of those systems whose capability pentagons contain the requirement pentagon.

The four systems feature different internal data representations and platforms for data operations. BigDAWG follows the thin-middleware principle and employs no internal data model or intermediate algebra for query translation and data transformation. In contrast, CloudMdsQL and Myria each have built-in relational data representation with a relational-algebra compiler. Further, the approach of Apache Drill is to adopt a JSON-based data model, to achieve more flexibility in processing raw data.

Evaluating systems with such diverse emphases and architectures is challenging in practice. BigDAWG aims at data integration with a fully federated architecture over collections of vertically integrated database engines. CloudMdsQL targets full expressiveness in the native queries for each database engine, while Myria focuses on an extended relational algebra for semantic equivalences between different data models. Apache Drill is designed for interactive data analysis over large-scale unstructured and weakly structured data. These different emphases determine the choices for the respective architectures, as it is perhaps overly optimistic to expect a single integration system to fit all possible workloads for a wide range of new and existing markets.

V. Related Work

As it became clear that a variety of data-processing architectures may be required for specialized markets, several directions were proposed for future data-management

Evaluation Framework		BigDAWG	CloudMdsQL	Myria	Apache Drill
Heterogeneity	Data-Store Heterogeneity	PostgreSQL, Apache Accumulo, and SciDB; potentially any data store	Sparksee, Derby, MongoDB, HDFS; potentially any data store	MyriaX, Radish, SciDB, HDFS, SPARQL; other data stores that can be handled by the RACO compiler	HDFS, HIVE, HBase, S3, MongoDB, Swift, and relational databases; limited support for graph, array, or stream data stores
	Processing-Engine Heterogeneity	relation, array, graph, stream, text, and other processing engines wrapped in information islands	component data store's processing engines; a relational engine and an Apache Spark engine for processing intermediate results	component data store's processing engines; extended relational and restricted Python engine	parallel SQL processing engine
	Query-Interface Heterogeneity	semantics in each information island's data model; one or more query interfaces with each information island	standard SQL with embedded functional subqueries; subqueries passed to the native query interfaces of the component data stores	MyriaL, an extended relational query interface and Python API	SQL, DrQL, MongoDB Query Language, and domain-specific languages that can be parsed to generate logical plans
Autonomy	Association Autonomy	associations via shims; catalog information updated automatically or maintained by administrators	association via wrappers; global catalog updated automatically or maintained by administrators	need to register each data store in RACO, such as adding rewrite rules	easy association with storage plugins
	Execution Autonomy	translation to native API calls, execution with no interference from the integration system	translation to native API calls, execution with no interference from the integration system	translation to native API calls, execution with no interference from the integration system	mostly scanning data into tables and executing queries by the Drill query engine; no interference into component data stores
	Evolution Autonomy	monitor updating catalog information to reflect evolutionary changes	changes handled by explicit queries	immutable data types; updating rewrite rules to accommodate evolutionary changes	recompiling from scratch with schema discovery
Transparency	Location Transparency	transparency within islands, need to specify information islands with **SCOPE** operators	need to specify data stores hosting the data	datasets stored as variables in MyriaX, need to specify the data store for ingestion	need to specify data stores hosting the data
	Transformation Transparency	hiding transformation details with **CAST** operators	need to specify mappings between data types; automatic transformation to table representation	transformation managed by RACO and transparent to users	transparent with dynamic schema discovery
Flexibility	Schema Flexibility	schemata are expressed in the semantic context of information islands; no support for dynamic schema discovery	defining schemata in queries; no support for dynamic schema discovery	defining schemata via customizable scan functions	dynamic schema discovery
	Interface Flexibility	query interfaces of the fixed islands	allowing embedded functional subqueries and user-defined MFR functions	allowing user-defined functions and aggregates, constrained imperative constructs, and customized operators	allowing user-defined functions
	Architectural Flexibility	modularization into optimizer, monitor, executor, and migrator; not readily extensible	modularization into planner, executor, and wrapper; not readily extensible	extensibility to new data stores and optimizers by adding rewrite rules to RACO	supporting pluggable query language interfaces, query optimizers, and storage plugins
Optimality	Federated Plan Optimization	query rewriting via semantic equivalence and active data transformation and migration	pushing down subquery execution; no active transformation or migration	pushing down subquery execution; actively transforming and migrating data across data stores	pushing down subquery execution; no active transformation or migration
	Data-Placement Optimization	allowing data migration and replication	staging and caching frequent data sets in the table store	allowing data migration and replication	little support

Table I
SUMMARY OF REFERENCE SYSTEMS WITH THE PROPOSED EVALUATION FRAMEWORK.

systems [15], including multiple systems united by a common parser, multiple systems using abstract data types, data federation, and from-scratch rewrites. The proposed evaluation framework is largely inspired by early efforts on federated database systems [33], as heterogeneity, autonomy, and distributed processing are still concerns for modern integration systems. Meanwhile, semantic heterogeneity remains one of the biggest challenges in query processing over heterogeneous data, calling for better protocols and algorithms alongside new system designs. Most of the work discussed in this survey can be viewed as a generalization of the concept of "data federation." A framework that is similar to ours in spirit has been recently proposed in [43] for query processing in cloud multi-store systems.

VI. Conclusion

For all but the simplest problems, data are diverse. Forcing diverse data to fit into a single data model could lead to a host of problems ranging from greater integration complexity to reduced performance. Systems that integrate data but support multiple data stores have begun to emerge. We believe that more integration systems developed over time will become an important strand of Big Data Research.

To support this emerging research trend, it is important to understand integration systems relative to a systematic comparison framework. A major contribution of this paper is in proposing such a framework. Our framework defines a high-level taxonomy (federated, polyglot, multistore, and polystore systems) and a collection of qualitative evaluation criteria (heterogeneity, autonomy, transparency, flexibility and optimality). We consider four different systems with respect to our framework; BigDAWG, CloudMdsQL, Myria, and Apache Drill. Our results do not show that any one system is "better" than the others. Rather, the analysis brings the features of the different systems into sharp relief and helps clarify the design tradeoffs made in creating these systems. Furthermore, our analysis suggests important directions for future work on integration systems, in particular, establishing principles for cross-model query equivalence and containment, efficient data transformation (including migration between data stores), performance monitoring and automatic load balancing, and distributed locking and transaction management.

Acknowledgment

The authors would like to thank Paul Jones and Yuxu Yang for their valuable suggestions.

References

[1] A. E. Johnson, T. J. Pollard, L. Shen, L.-w. H. Lehman, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L. A. Celi, and R. G. Mark, "MIMIC-III, a freely accessible critical care database," *Scientific Data*, vol. 3, 2016.

[2] 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*, vol. 8, no. 12, pp. 1908–1911, 2015.

[3] T. Mattson, V. Gadepally, Z. She, A. Dziedzic, and J. Parkhurst, "Demonstrating the BigDAWG polystore system for ocean metagenomic analysis," in *Proc. Conference on Innovative Data Systems Research (CIDR'17)*, 2017, pp. 1–9.

[4] F. Bugiotti, D. Bursztyn, A. Deutsch, I. Ileana, and I. Manolescu, "Invisible glue: Scalable self-tuning multi-stores," in *Proc. Conference on Innovative Data Systems Research (CIDR'15)*, 2015.

[5] R. Bonaque, T. D. Cao, B. Cautis, F. Goasdoué, J. Letelier, I. Manolescu, O. Mendoza, S. Ribeiro, and X. Tannier, "Mixed-instance querying: a lightweight integration architecture for data journalism," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1513–1516, 2016.

[6] M. Stonebraker, "The case for polystores," `http://wp.sigmod.org/?p=1629`, Jul 2015.

[7] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik, "The BigDAWG polystore system," *ACM SIGMOD Record*, vol. 44, no. 2, pp. 11–16, 2015.

[8] V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker, "The BigDAWG polystore system and architecture," in *Proc. IEEE High Performance Extreme Computing Conference (HPEC'16)*, 2016, pp. 1–6.

[9] B. Kolev, C. Bondiombouy, O. Levchenko, P. Valduriez, R. Jimenez-Péris, R. Pau, and J. Pereira, "Design and implementation of the CloudMdsQL multistore system," in *Proc. Cloud Computing and Services Science (CLOSER'16)*, vol. 1, 2016, pp. 352–359.

[10] B. Kolev, C. Bondiombouy, P. Valduriez, R. Jiménez-Peris, R. Pau, and J. Pereira, "The CloudMdsQL multistore system," in *Proc. ACM International Conference on Management of Data (SIGMOD'16)*, 2016, pp. 2113–2116.

[11] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta *et al.*, "The Myria big data management and analytics system and cloud service," 2017.

[12] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker *et al.*, "Demonstration of the Myria big data management service," in *Proc. ACM International Conference on Management of Data (SIGMOD'14)*, 2014, pp. 881–884.

[13] M. Hausenblas and J. Nadeau, "Apache Drill: Interactive ad-hoc analysis at scale," *Big Data*, vol. 1, no. 2, pp. 100–104, 2013.

[14] M. Stonebraker and U. Çetintemel, "One size fits all: an idea whose time has come and gone," in *Proc. IEEE International Conference on Data Engineering (ICDE'05)*, 2005, pp. 2–11.

[15] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik, "One size fits all? part 2: Benchmarking results," in *Proc. Conference on Innovative Data Systems Research (CIDR'07)*, 2007.

[16] J. Widom, "Research problems in data warehousing," in

Proc. ACM International Conference on Information and Knowledge Management (CIKM'95), 1995, pp. 25–30.

[17] J. M. Smith, P. A. Bernstein, U. Dayal, N. Goodman, T. Landers, K. W. Lin, and E. Wong, "Multibase: Integrating heterogeneous distributed database systems," in *Proc. National Computer Conference (NCC'81)*. ACM, 1981, pp. 487–499.

[18] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark SQL: Relational data processing in Spark," in *Proc. ACM International Conference on Management of Data (SIGMOD'15)*, 2015, pp. 1383–1394.

[19] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 922–933, 2009.

[20] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling, "Split query processing in Polybase," in *Proc. ACM International Conference on Management of Data (SIGMOD'13)*, 2013, pp. 1255–1266.

[21] Y. Tian, T. Zou, F. Ozcan, R. Goncalves, and H. Pirahesh, "Joins for hybrid warehouses: Exploiting massive parallelism in Hadoop and enterprise data warehouses." in *Proc. International Conference on Extending Database Technology (EDBT'15)*, 2015, pp. 373–384.

[22] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a Map-Reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[23] M. Zhu and T. Risch, "Querying combined cloud-based and relational databases," in *Proc. IEEE International Conference on Cloud and Service Computing (CSC'11)*, 2011, pp. 330–335.

[24] K. W. Ong, Y. Papakonstantinou, and R. Vernoux, "The SQL++ unifying semi-structured query language, and an expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewSQL databases," *CoRR, abs/1405.3631*, 2014.

[25] J. Kepner, W. Arcand, W. Bergeron, N. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz *et al.*, "Dynamic distributed dimensional data model (D4M) database and computation system," in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'12)*, 2012, pp. 5349–5352.

[26] H. Hacígümüş, J. Sankaranarayanan, J. Tatemura, J. LeFevre, and N. Polyzotis, "Odyssey: a multistore system for evolutionary analytics," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1180–1181, 2013.

[27] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey, "MISO: Souping up big data query processing with a multistore system," in *Proc. ACM International Conference on Management of Data (SIGMOD'14)*, 2014, pp. 1591–1602.

[28] E. Kharlamov, T. Mailis, K. Bereta, D. Bilidas, S. Brandt, E. Jimenez-Ruiz, S. Lamparter, C. Neuenstadt, O. Özçep, A. Soylu *et al.*, "A semantic approach to polystores," in *Proc. IEEE International Conference on Big Data (ICBD'16)*, 2016, pp. 2565–2573.

[29] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal, "Optimizing analytic data flows for multiple execution engines," in *Proc. ACM International Conference on Management of Data (SIGMOD'12)*, 2012, pp. 829–840.

[30] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand, "Musketeer: all for one, one for all in data processing systems," in *Proc. ACM European Conference on Computer Systems (Eurosys'15)*, 2015, p. 2.

[31] D. Agrawal, L. Ba, L. Berti-Equille, S. Chawla, A. Elmagarmid, H. Hammady, Y. Idris, Z. Kaoudi, Z. Khayyat, S. Kruse *et al.*, "Rheem: Enabling multi-platform task execution," in *Proc. ACM International Conference on Management of Data (SIGMOD'16)*, 2016, pp. 2069–2072.

[32] S. Dasgupta, K. Coakley, and A. Gupta, "Analytics-driven data ingestion and derivation in the AWESOME polystore," in *Proc. IEEE International Conference on Big Data (ICBD'16)*, 2016, pp. 2555–2564.

[33] A. P. Sheth and J. A. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Computing Surveys (CSUR)*, vol. 22, no. 3, pp. 183–236, 1990.

[34] P. Bailis, J. M. Hellerstein, and M. Stonebraker, "Readings in database systems," `http://www.redbook.io/`, 2015.

[35] Z. She, S. Ravishankar, and J. Duggan, "BigDAWG polystore query optimization through semantic equivalences," in *Proc. IEEE High Performance Extreme Computing Conference (HPEC'16)*, 2016, pp. 1–6.

[36] P. Chen, V. Gadepally, and M. Stonebraker, "The BigDAWG monitoring framework," in *Proc. IEEE High Performance Extreme Computing Conference (HPEC'16)*, 2016, pp. 1–6.

[37] A. Dziedzic, A. J. Elmore, and M. Stonebraker, "Data transformation and migration in polystores," in *Proc. IEEE High Performance Extreme Computing Conference (HPEC'16)*, 2016, pp. 1–6.

[38] A. M. Gupta, V. Gadepally, and M. Stonebraker, "Cross-engine query execution in federated database systems," in *Proc. IEEE High Performance Extreme Computing Conference (HPEC'16)*, 2016, pp. 1–6.

[39] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. ACM Symposium on Cloud Computing (SoCC'13)*, 2013, p. 5.

[40] B. Haynes, A. Cheung, and M. Balazinska, "PipeGen: Data pipe generator for hybrid analytics," *arXiv preprint arXiv:1605.01664*, 2016.

[41] R. Moffatt, "How to guide: Getting started with Apache Drill," `https://mapr.com/blog/how-guide-getting-started-apache-drill/`, Sept 2016.

[42] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems." in *Proc. USENIX Annual Technical Conference (ATC'17)*, vol. 8, 2010, p. 9.

[43] C. Bondiombouy and P. Valduriez, "Query processing in multistore systems: an overview," *International Journal of Cloud Computing*, vol. 5, no. 4, pp. 309–346, 2016.