QueryArrow: Semantically Unified Query and Update of Heterogeneous Data Stores

Hao Xu
University of North Carolina at Chapel Hill
xuhao@renci.org
Ben Keller
University of North Carolina at Chapel Hill
kellerb@renci.org
Antoine de Torcy
University of North Carolina at Chapel Hill
adtorcy@renci.org
Jason Coposky
University of North Carolina at Chapel Hill
jasonc@renci.org

ABSTRACT

Modern system applications often need to interact with metadata from multiple, heterogeneous data stores. An ad hoc solution for integration of multiple data stores by issuing individual statements in the languages of the data stores runs the risk of semantic incompatibilities. This paper describes QueryArrow, a generic software that provides a semantically unified query and update interface to multiple types of data stores. QueryArrow has an algebra-based language called QueryArrow Language (QAL), which can be partially translated to languages of different data stores. We describe the design of QueryArrow, the syntax and semantics of QAL, how QAL is translated to languages of different data stores, and demonstrates its applications as an iRODS database plugin.

1. INTRODUCTION

Modern system applications often need to interact with data from multiple, heterogeneous data stores. There are several recurring tasks in such applications, including the aggregation, access control, discovery, and migration of metadata. A software solution for this challenge is tricky because of the diverse range of types of data stores that it must interact with, including, for example, relational databases, graph databases, and document-oriented databases. Different types of data stores have different types of query languages and data manipulation languages, different semantics of the languages, and different levels of capabilities (such as support for features such as regular expressions).

Some existing solutions aim to bridge the gap between these data store, by either creating a unified query language or API, such as SQL++ [9] and UnQL [6], or refitting a current popular query language or API designed for one type of database into other types of databases, such as Presto [1] and Spark SQL [2]. However, there are at least one of the following two drawbacks in the current solutions:

- The solution does not have a formal definition of the semantics of their query language. Therefore, an ad hoc solution, where results are aggregated from multiple data stores by issuing an individual query in the query language of each database, runs the risk of semantic incompatibilities.
- The solution is query-only and lacks bidirectional support for both query and update. Without support to update, we cannot provide an abstraction of data stores that makes data access transparent to client applications where the metadata is mutable.

iRODS UGM 2017 June 13-15, 2017, Utrecht, Netherlands

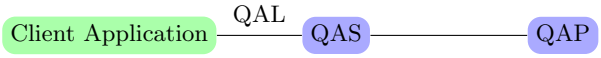


Figure 1: Architecture Diagram

A more principled approach would require complete semantic decoupling of client API from underlying data stores where data are allowed to be mutable. This allows migrating the underlying data stores without modification to the client application. This requires a unified query and data manipulation language. Such a language inevitably is a superset of capabilities of actual data store languages. Therefore, we need to integrate the notion of partially supported features and schemata into this framework, so that we can combine multiple data stores with different capabilities into a unified data store which provides full support to the features and schemata required by the client application.

QueryArrow [14, 16] is built on theoretical developments that allow us to create software that is based on rigorous treatment of semantic foundations of a unified query and data manipulation language. Categorical models of databases allow us to represent the same information in both relational databases and graph databases and transform between the two different representations [13], which is used in QueryArrow for automatic generation of graph database translators. QueryArrow Language (QAL) is structured as near-semiring [8], which allows us to model both query and update, and provide general laws for optimization. QueryArrow is similar to Transaction Logic [5], but we give QAL a monadic semantics [4], which is formalized in Coq [7].

This paper introduces the design of QueryArrow, the syntax and semantics of QAL, how QAL is translated to different database languages, and demonstrates its applications.

2. DESIGN OF QUERYARROW

QueryArrow is made up of three elements: the QueryArrow Service, the QueryArrow Language, and the QueryArrow Plugins (QAP), as shown in Figure 1.

- QueryArrow Service: Register QAP and support execution of QAL
- QueryArrow Language: Provide a semantically unified configuration language, query language, and data manipulation language.
- QueryArrow Plugins: Provide mappings between QAL and external data stores.

A QueryArrow instance includes a QueryArrow Service and a composition of QAPs.

There are three types of QAPs. A data store QAP interfaces with a data store by translating QAL to the language of the data store and interpreting the results returned by the data store. A meta QAP allows users to add aggregation, policy and distributed support to our architecture without additional complexity to the core codebase. An in-memory QAP provides various in-memory functionalities. Currently supported QAPs are shown in the Table 1.

A translation QAP enables translation of the QAL back into QAL, according to user defined rewriting rules written in the configuration fragment of QAL. This allows user to define policy on their metadata. This enables the definition of policies such as metadata access control, distribution, and retrieval optimization. A typical QAP composition is show in Figure 2.

QueryArrow can be run in a distributed environment. A remote QAP is implemented to allow using a QAS as a data store.

3. SYNTAX AND SEMANTICS OF QAL

3.1 Syntax

Name	Description
Sum QAP	aggregation
Translation QAP	policy support
Cache QAP	caching
QAS QAP	remoting
Mutable Map QAP	in-memory mutable map
Immutable Map QAP	in-memory immutable map
ElasticSearch QAP	interfacing with ElasticSearch
Neo4j QAP	interfacing with Neo4j
PostgreSQL QAP	interfacing with Postgres
SQLite3 QAP	interfacing with SQLite3
CockroachDB QAP	interfacing with CockroachDB
FileSystem QAP	file system

Table 1: Available QAPs

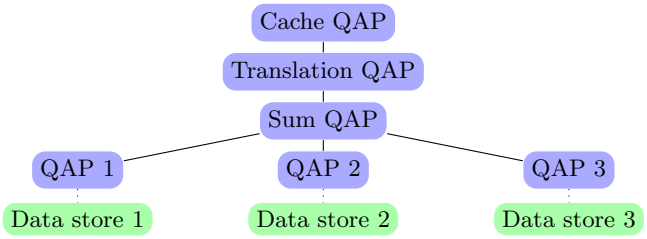


Figure 2: Example QAP Composition

p		*literal*	
$prty$		*primitive type*	
v		*variable*	
P		*primitive predicate name*	
N		*namespace*	
QP	::=	$P \mid N.QP$	*predicate name*
ty	::=	$prty \mid \texttt{list}\ ty$	*types*
t	::=	$p \mid v \mid [t_1, \ldots, t_n] \mid ty\ t$	*terms*
a	::=	$QP(t_1, \ldots, t_n)$	*atom*
c	::=	$a \mid \texttt{insert}\ a \mid \texttt{delete}\ a \mid g\ c \mid b$	
		$\mid \mathbf{1} \mid \mathbf{0} \mid c \oplus c \mid c \otimes c$	*command*
g	::=	$\neg \mid \exists \mid \texttt{distinct} \mid \texttt{limit}\ n \mid \texttt{order by}\ (\texttt{asc} \mid \texttt{desc})$	
		$\mid \texttt{let}\ v_1 = s_1, \ldots, v_n = s_n\ (\texttt{group by}\ v_1 \ldots v_n)?$	*aggregation*
s	::=	$\texttt{max}\ v \mid \texttt{min}\ v \mid \texttt{average}\ v \mid \texttt{sum}\ v \mid \texttt{count} \mid \texttt{count distinct}\ v \mid \texttt{some}\ v$	*summary*
pt	::=	$\texttt{input}?\ \texttt{output}?\ \texttt{key}?\ ty$	
R	::=	$\texttt{rewrite}\ (a \mid \texttt{insert}\ a \mid \texttt{delete}\ a)\ c$	
		$\mid \texttt{predicate}\ P(pt_1, \ldots, pt_n)$	
		$\mid \texttt{import qualified}?\ (\texttt{all} \mid P_1, \ldots, P_n \mid \texttt{all except}\ P_1, \ldots, P_n)\ \texttt{from}\ N$	
		$\mid \texttt{export}\ P_1, \ldots, P_n$	
		$\mid \texttt{export qualified}?\ (\texttt{all} \mid P_1, \ldots, P_n \mid \texttt{all except}\ P_1, \ldots, P_n)\ \texttt{from}\ N$	*configuration*
$prog$	::=	$R_1 \ldots R_n$	*program*

Figure 3: QAL Syntax

The syntax is given in Figure 3. A term t is either a literal p, a variable v, a list of terms, or a type coercion. An atom a is $QP(t_1, \ldots, t_n)$ where QP is a predicate name. c includes four primitive commands: a query command a, an insert command *insert* a, a delete command *delete* a, and an aggregation command $g\ a$, where g is functions such as max, min, average, sum, count, and count distinct, limit results to first n results, order results by ascending order, order results by descending order, return distinct results, test that results does not exist, test that results exists, or keep certain columns in the results. A command c is either a primitive command or one of the four composite commands: skip $\mathbf{1}$, stop $\mathbf{0}$, choice $c \oplus c$, sequencing $c \otimes c$.

In addition to commands, QAL also allows declaring new predicates and specifying rewriting rules, which are essential for defining policies. The rewriting rules allows us to rewrite a query command, an insert command, or a delete command to arbitrary commands. Also, we have import and export statements. The details of these declarations are given in [16].

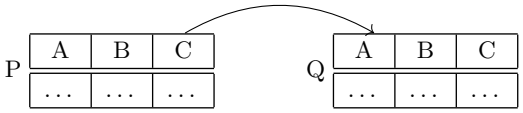
Examples applications of QAL are described in Section 5.

3.2 Semantics

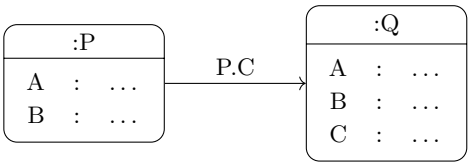
The abstract semantics of QAL is formally specified in [15] using Coq. In this subsection, we give a brief description of the formalization.

The key to the formalization is specifying various Haskell typeclasses including `Functor`, `Applicative`, `Monad`, `Traversable`, `Alternative`, `Foldable`, and `Monoid`. We also defined their instances. In addition, we specified `NearSemiRing`. We model all of these in terms of setoids.

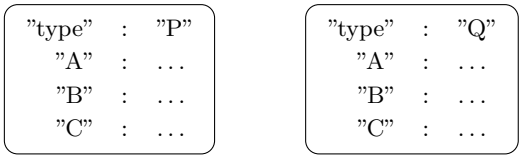
The semantics of commands are given in a "store-and-heap" monad, written `sh`, which is a specialization of the `ContT` monad transformer. We define a semantic equivalence relation between commands in terms of this semantics. We have proved that not only is `sh` a monad, but also that QAL forms a near-semiring $(\mathcal{L}(c), \mathbf{0}, \mathbf{1}, \oplus, \otimes)$, where $\mathcal{L}(c)$ is



(a) Example SQL Tables



(b) Example Cypher Graph



(c) Example ElasticSearch Documents

Figure 4: Examples

the language generated by c. This allows us to give an interpretation of $\mathbf{0}$, $\mathbf{1}$, $\oplus$, and $\otimes$. Other language constructs are interpreted in the monadic instance of `sh`.

One challenge to the applicability of this abstract semantics is how to incorporate concrete semantics of different data stores. Inevitably some QAL commands cannot be translated to one statement of one data store. In this case, the QAL command is partially translated, and each subexpression that can be translated into one statement of one data store is executed, and the non-translatable part are executed by a generic execution function. To illustrate this, consider the following example. Suppose that we have three data stores, exporting predicates P, Q, and R, respectively. When a user issues a command such as $P(x, a) \otimes (Q(x, y) \oplus R(x, y))$, the predicates are translated into the languages of the respective data stores, dispatched, and the results are collected and combined according to the abstract semantics. How do we integrate the semantics of different data stores, thereby giving semantics to the whole command? The solution is using built-in commands: we can specify statements in the languages of data stores as built-in commands and plug them into the abstract semantics, as long as they can be interpreted in the `sh` monad. For each language $\mathcal{L}$ of a data stores, a translator from QAL then is a partial function from commands to that language $trans : \mathcal{L}(c) \to \mathcal{L}$. The `sh` monad is designed such that it is parametric to module types `AbstractStore` and `AbstractHeap`. This way, we can choose different concrete implementations of these data structures for different combinations of data stores. An ongoing effort is to integrate SQL into this abstract semantics.

4. TRANSLATION

In this section we list three types of data stores and how QAL is translated into their statements. Our goal is to translate the same commands into multiple data stores which are semantically equivalent based on the concept of observational equivalence: informally, given two data stores, define a relation R of related states of the data stores, if a translation of every successful command from QAL to the languages of the two data stores takes related states to related states, and returns the same set of results, then we say that under the translation, the two data stores are observationally equivalent. For a formal treatment of bisimulation, see [11]. In this section, we assume x, y, z are variables, and a, b, c are primitive values.

4.1 Relational Database

SQL is the query language and data manipulation language for relational databases. In the PostgreSQL QAP, a primitive command is translated to an SQL statement. Users are allowed to defined arbitrary predicates and

translations. In addition, QueryArrow provides an automatic translation generator based on a database schema to reduce the coding needed to create QAPs. We illustrate the translation in an example as show in Figure 4a, where $P.A$ and $Q.A$ are primaries keys and $P.C$ is a foreign key to $Q.A$. The automatic translator generates six predicates in two categories: object predicates $PA(x)$ and $QA(x)$, and property predicates $PB(x, y)$, $PC(x, y)$, $QB(x, y)$, and $QC(x, y)$. Object predicates and property predicates differ in how they are translated in insert and delete commands. Object predicates are translated to `INSERT` and `DELETE` statement. Property predicates are translated to `UPDATE` statements. For example, we translate PB as follows:

- query $PB(a, x)$: `SELECT B FROM P WHERE A = a`
- insert $insert\ PB(a, b)$: `UPDATE P SET B = b WHERE A = a`
- delete $delete\ PB(a, b)$: `UPDATE P SET B = NULL WHERE A = a`

and QA as follows:

- query $QA(x)$: `SELECT A FROM Q`
- insert $insert\ QA(a)$: `INSERT INTO Q (A) VALUES (a)`
- delete $delete\ QA(a)$: `DELETE FROM Q WHERE A = a`

Composite commands are translated as nullary and binary partial functions that combine SQL queries. This allows us to, for example, insert a table with columns with `NOT NULL` constraints. For example, if $P.B$ is not null, then the translation of $insert\ PA(a)$,

```
INSERT INTO P (A) VALUES (a)
```

is not valid SQL insert. And $insert\ PB(a, b)$ is translated to

```
UPDATE P SET B = b WHERE A = a
```

But $insert\ PA(a) \otimes insert\ PB(a, b)$ should be translated to

```
INSERT INTO P (A,B) VALUES (a,b)
```

which is valid.

4.2 Graph Database

Cypher is the query language and data manipulation language for Neo4j [3]. In the Neo4j QAP, a primitive command is translated to an Cypher statement. A graph schema is automatically generated by a translation generator from a SQL schema by QueryArrow as follows, following a variation of the mapping given in [12]: Each table is translated into a node and each column is translated into one of the three: A primary key is translated into a property of the node. A foreign key is translated into an edge. Other columns are translated into a property. Special clauses are added to the translation of insert commands to ensure that primary keys are unique. For example, given SQL table schema as shown in Figure 4a, we can generate a model as shown in Figure 4b.

We may translate PB as follows:

- query $PB(a, x)$: `MATCH (n:P) WHERE n.A = a RETURN n.B`
- insert $insert\ PB(a, b)$: `MATCH (n:P) WHERE n.A = a SET n.B = b`

- delete *delete* $PB(a, b)$: `MATCH (n:P) WHERE n.A = a SET n.B = NULL`

and QA as follows:

- query $QA(x)$: `MATCH (n:Q) RETURN n.A`
- insert *insert* $QA(a)$: `MERGE (n:Q{A:a})`
- delete *delete* $QA(a)$: `MATCH (n:Q) WHERE n.A = a DELETE n`

One subtle issue is failure modes. The semantics of relational database and graph database usually do not match. For example, in some graph databases, users are not able to specified unique properties or required properties. Therefore, a translation of *insert* $PA(a)$ is a valid statement, even if the key value a already exists. There are currently two solutions. We can simulate. For example, in the unique property case, we can simulate using the `ON CREATE` clause. We can also create an abstraction layer in which both types of databases implement the same semantics. For example, in the required property case, we can create a predicate $P(x, y, z)$ and hide the more primitive $PA(x)$. This can be done using rewriting rules, import, and export features of the QAL.

4.3 Document-oriented Database

ElasticSearch provides a JSON-based query language and data manipulation language which are radically different from traditional databases. In particular, ElasticSearch is an example of Document-oriented Database. In this type of databases, each predicate is naturally translated to a partial document. For example, given SQL table schema as show in Figure 4a, we can generate a model as show in Figure 4c. This model, unlike the Cypher's case, is often weaker than the SQL model, because it doesn't explicitly encode relations. The translation of predicates is similar to that of graph databases. Because of lack of operators like SQL's `JOIN` or `UNION`, the translation function is mostly undefined on commands with nontrivial combinations of composite commands.

5. APPLICATION EXAMPLES

iRODS QueryArrow database plugin enables iRODS [10] to use QueryArrow as iCAT. This enables the following application examples.

Metadata Access Control. iRODS allows users to tag data object with metadata in the forms (attribute, value, unit) triples. The data management solution stores such metadata in a relational database and is not designed with metadata access control. QueryArrow allows us to add metadata access control using QueryArrow, by defining rewriting rules, without changing the schema of the original database. The extra information such as access control list (ACL) is stored an external database and integrated into the application by QueryArrow.

Metadata Migration. In iRODS, metadata are stored in a relational database. QueryArrow allows us to migrate part of the metadata into a graph database.

Metadata Indexing. As the number of data objects grows, regular queries for data objects become slow. QueryArrow allows us to write rewriting rules so that some of the metadata are put into a search engine based on their attribute name. When user add or remove a metadata item with an indexed attribute name, it is added or removed to the search engine. When the user queries data object by those metadata attribute names, the search engine is utilized accelerate the query.

All of this is transparent to the client application.

6. SUMMARY

In this paper, we introduced design of QueryArrow, the syntax and semantics of QAL, how QAL is translated, and its application examples in iRODS.

REFERENCES

[1] `https://prestodb.io/`.

[2] `http://spark.apache.org/sql/`.

[3] `https://neo4j.com/`.

[4] *Computational Lambda-Calculus and Monads*, 1989.

[5] Anthony J. Bonner and Michael Kifer. Transaction logic programming. In *International Conference on Logic Programming*, pages 257–279, 1993.

[6] Peter Buneman, Mary F. Fernandez, and Dan Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB Journal: Very Large Data Bases*, 9(1):76–110, ???? 2000.

[7] Adam Chlipala. *Certified Programming with Dependent Types : A Pragmatic Introduction to the Coq Proof Assistant*. 2013.

[8] Jules Desharnais and Georg Struth. Domain axioms for a family of Near-Semirings. In José Meseguer and Grigore Roşu, editors, *Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 330–345. Springer Berlin Heidelberg, 2008.

[9] Kian W. Ong, Yannis Papakonstantinou, and Romain Vernoux. The SQL++ query language: Configurable, unifying and semi-structured, December 2015.

[10] Arcot Rajasekar, Reagan Moore, Chien-Yi Hou, Christopher A. Lee, Richard Marciano, Antoine de Torcy, Michael Wan, Wayne Schroeder, Sheau-Yen Chen, Lucas Gilbert, Paul Tooby, and Bing Zhu. iRODS primer: Integrated rule-oriented data system. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 2(1):1–143, January 2010.

[11] J. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, October 2000.

[12] David I. Spivak. Simplicial databases. April 2009.

[13] David I. Spivak. Functorial data migration. *Information and Computation*, May 2012.

[14] Hao Xu. `https://github.com/xu-hao/QueryArrow`, 2017.

[15] Hao Xu. `https://github.com/xu-hao/CertifiedQueryArrow`, 2017.

[16] Hao Xu, Ben Keller, Antoine de Torcy, and Jason Coposky. Queryarrow: Bidirectional integration of multiple metadata sources. *8th iRODS User Group Meeting, University of North Carolina at Chapel Hill*, June 2016.