Technical Report: QueryArrow: Bidirectional Integration of Multiple Metadata Sources

Hao Xu[1][2]
xuh@email.unc.edu
University of North Carolina at Chapel Hill

in collaboration with

Ben Keller Antoine de Torcy Jason Coposky
iRODS Consortium

Abstract

This paper describes QueryArrow, a generic software that provides a semantically unified query and update interface to a wide range of metadata sources.

Introduction

The "Big metadata" challenge is analogous to the "Big data" challenge. Examples include:

- Aggregating metadata stored in multiple, heterogeneous databases
- Managing access control of metadata items
- Metadata based indexing
- Metadata migration

A software solution for the "Big metadata" challenge is trickier compared to the "Big data" challenge in the diverse range of types of data sources that it must interact with, including, for example, relational databases, graph databases, streams databases, and general web services. Different types of data sources have different types of semantics, and different levels of capability, and support for features such as regular expressions.

[1] This research is partially supported by the National Science Foundation under Grant Number OCI 0940841 DataNet Federation Consortium. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.
[2] This research is partially supported by iRODS Consortium.

Table 1 Semantics of Query Results

	duplication	ordering	infinite data
finite set	no	no	no
finite multiset	yes	no	no
list	yes	yes	no
stream	yes	yes	yes

Even within SQL-like queries, one can have multiple semantics such as set, multiset, list, or stream, as summarized in Table 1. For example traditional SQL is based on a combination of list, multiset, and set semantics, whereas a stream database can only be based on stream semantics because of the potentially infinite nature of streams.

Many commercial or open source databases do not have a formal definition of the semantics of their query language. Therefore, an ad hoc solution, where results are aggregated from multiple metadata sources by issuing an individual query in the query language of each database, runs the risk of semantic incompatibilities. For example, if one database uses list semantics, while another uses set semantics, the aggregated result will be incompatible. The ad hoc approach is also prone to error caused by semantic changes when upgrading to a new version of a database.

Therefore, a strategy to mitigate this challenge is to formally define the semantics that domain applications expect and create a software middleware to translate queries and results between the domain applications and the databases. This allows us to tackle big metadata challenges as follows:

- For aggregating metadata stored in multiple, heterogeneous databases, we can issue all queries in a unified query language with formally defined semantics, ensuring that the results are compatible.
- For managing access control of metadata items, we can define in the translation step additional access control mechanisms, without requiring that the underlying database support such mechanisms.
- For metadata based indexing, we can define the translation between traditional databases and search engines.
- For metadata migration, when the semantics of the underlying database changes, we can redefine the translation without changing the domain application.

QueryArrow

Design Elements

The goal of QueryArrow is to provide generic software middleware that provides a semantically unified query and update interface to a wide range of metadata sources. To achieve this goal, QueryArrow must be generic:

- It must be representation independent. We distinguish how metadata is represented in the database, such as tables, graphs, etc. from the information content of the metadata. A central idea of QueryArrow is that we allow the information content to be utilized and migrated independently from how it is represented.
- It must be configurable. Many metadata policies depend on the application domain. In order for a QueryArrow instance to conform to such policies, it must be configurable.

When integrated with data grid middleware such as the integrated Rule Oriented Data System (iRODS), QueryArrow removes the current iRODS limitation of one relational database per zone for metadata query and update capabilities.[3]

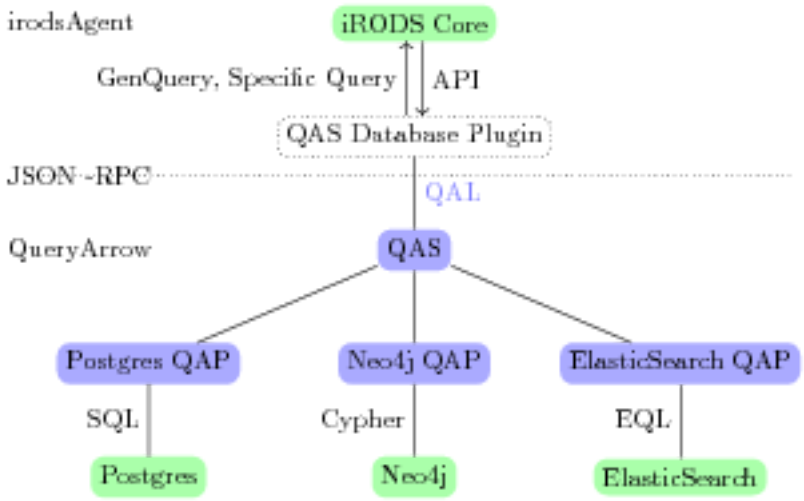

Figure 1 QueryArrow Architecture Diagram

[3] Although other databases can be queried within iRODS via rules and microservices, there is no systematic way for combining the results which provides consistency guarantees.

QueryArrow is made up of three elements: the QueryArrow Service, the QueryArrow Language, and the QueryArrow Plugins. The iRODS data grid utilizes QueryArrow via a JSON-RPC mechanism, as shown in Figure 1.

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

A QueryArrow instance includes a QueryArrow Service and QueryArrow plugins. Each plugin provides an interface with one database. QueryArrow communicates with service consumers via JSON-RPC. The queries are issued from the client in the QueryArrow Language and can be translated into both SQL and NoSQL query languages. Each QueryArrow plugin translates queries to its database.

Currently, QueryArrow implements generic plugins for Postgres, Neo4j, SQLite, CockroachDB (via Postgres API), and a domain specific plugin for ElasticSearch.

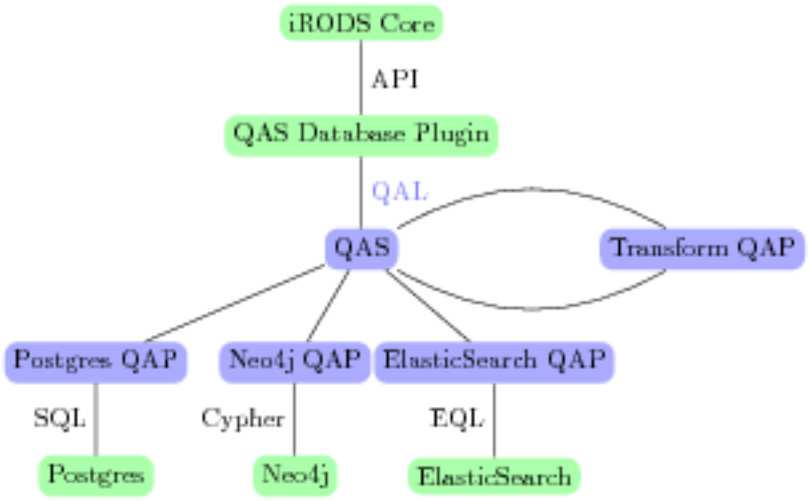

Figure 2 Policy Support

In addition, a transformation QAP plugin is implemented, as shown in Figure 2. This QueryArrow plugin enables translation of the QueryArrow Language back into the QueryArrow Language. The configuration fragment of QAL allows users to define customized rules on their metadata and data. In particular, QueryArrow supports the imposition of constraint policies. This enables the definition of policies such as metadata access control, distribution, and retrieval optimization..

QAL

The following concerns about SQL are the main motivations for creating QAL:

- SQL is strong in query support but weak in data manipulation.
- SQL performance is dependent on an individual DB's query optimizer. When migrating data between different databases, there is a need to craft different SQL for different databases to achieve optimal performance.
- SQL doesn't support the notion of multiple databases.
- SQL has limited, unidirectional support for transforming queries via views (needed for applying policies).
- SQL cannot be easily translated to other database paradigms.

QAL is based on theoretical results from the research community. It is based on features from Relational Algebra, Process Algebra, and Substructural Logic. QAL takes the core features of SQL based on relational algebra, and extends it with process algebra, to provide a language that can be translated into both SQL and NoSQL databases. QAL also supports translating different subexpressions of the same query to different query languages and combining them in a semantically consistent manner. In this section, we take a look at the syntax and information semantics of QueryArrow and the formalization of QAL, and how it guarantees semantic consistency.

Syntax

N namespace, P predicate, i int, s string, v variable

t	::=	$i \mid s \mid v$	*terms*
a	::=	$N.P(t_1,\ldots,t_n) \mid P(t_1,\ldots,t_n)$	*atom*
c	::=	$a \mid insert\ a \mid delete\ a$	
		$\mid\sim c \mid exists\ c \mid let\ v = g,\ldots,v = g\ c \mid limit\ i\ c \mid order\ by\ v\ (asc \mid desc)$	
		$\mid one \mid zero \mid c \mid c \mid c\ c$	
		$\mid return\ v_1 \ldots v_n$	*command*
g	::=	$max\ v \mid min\ v \mid count$	*aggregation*
R	::=	$rewrite\ a\ c \mid rewrite\ insert\ a\ c$	
		$\mid rewrite\ delete\ a\ c$	*rewriting rules*
I	::=	$import\ qualified?\ (all \mid P_1,\ldots,P_n)\ from\ N$	*import*
E	::=	$export\ qualified?\ (all \mid P_1,\ldots,P_n)\ (from\ N)?$	*export*

Figure 3 Syntax of QAL

The QAL syntax is summarized in Figure 3.

Semantics

An informal description of the semantics of queries and updates is given as follows:

- a : a simple query

- *insert a* : a simple insert update
- *delete a* : a simple delete update
- $c_1 c_2$: run c_1, then filter the results by c_2. This corresponds to a conjunction in relational algebra, and a sequencing combinator in process algebra.
- $c_1 | c_2$: run c_1, c_2, then combine the results of the runs. This corresponds to a disjunction in relational algebra, and a choice combinator in process algebra.
- *zero* : empty results. This corresponds to false in relational algebra, and stop in process algebra.
- *one* : trivial query. This corresponds to true in relational algebra, and skip in process algebra.
- *~c* : run c. If the result is empty, then behave as *one*. Otherwise, behave as *zero* .
- *exists c* : run c. If the result is empty, then behave as *zero*. Otherwise, behave as *one* .
- *return* $v_1 \ldots v_n$. Project the results.
- *limit i c* : run c. Return first i results.
- *order by v* (*asc*|*desc*) . Sort results.
- *let* $v_1 = g_1, \ldots, v_n = g_n$ *c* . Run aggregate functions on results.

An informal semantics of configuration is given as follows:

- *rewrite a c* rewrite a as c.
- *rewrite insert a c* rewrite *insert a* as c.
- *rewrite delete a c* rewrite *delete a* as c.
- *import …* This imports predicates from other namespaces.
- *export …* This exports predicates to other namespaces.

Examples of QAL

In the examples, query expressions define how iRODS persistent state attributes are manipulated. Note that in iROdata_coll_id is the collection id that contains the data object.

Return all data objects ids and their names. The query expression finds the data_name for a given data_id and returns data_name, data_id.

```
DATA_NAME(x, y) return x y
```

Return all data objects names in collection `c`. The query expression finds the coll_id for a given collection name, finds the data_ids which have a data_coll_id = coll_id, and finds the data_names from the data_ids.

```
COLL_NAME(x, "c") DATA_COLL_ID(y, x) DATA_NAME(y, z)
return z
```

Return all data objects names in collection "c" or "c2". The query expression finds the coll_id for collection name "c", adds the coll_id for the collection name "c2", finds the data_ids which have a data_coll_id = coll_id, and finds the data_names from the data_ids.

```
(COLL_NAME(x, "c") | COLL_NAME(x, "c2"))
DATA_COLL_ID(y, x) DATA_NAME(y, z) return z
```

Return all data objects that do not belong to collection `c`. The query expression finds the data_ids that have a data_coll_id equal to a given coll_id, and finds the data_name from the data_id, where the coll_id is not the coll_id of a collection whose name is "c".

```
DATA_COLL_ID(y, x) DATA_NAME(y, z) ~COLL_NAME(x, "c")
return z
```

insert a new data object named `a`. The query expression creates a new data_id, and creates a new data object in database with the data_id and data_name.

```
nextid(x) insert DATA_OBJ(x) DATA_NAME(x, "a")
```

delete all data objects named `a`. The query expression finds the data_id corresponding to the data name "a", and deletes the entry for the data location.

```
DATA_NAME(x,"a") delete DATA_OBJ(x)
```

Translation examples of QAL

Return all data objects ids and their names.

```
DATA_NAME(x, y) return x y
```

SQL translation for iRODS data grid:

```
select data_id, data_name from r_data_main
```

Cypher translation:

```
match (var0:DataObject)
return var0.data_id, var0.data_name
```

ElasticSearch translation:

```
{
 "query":{
  "bool":{"must":[{"term":{"obj_type":"DataObject"}}]}
 }
}
```

QAL Rules

In contrast to existing iRODS rule engines, which allows defining procedural rules, QAL allows defining constraint rules. Whereas procedural rules are sufficient for tasks such as generating audit logs, constraint rules are necessary for tasks such as metadata access control, in which case simply logging that metadata access control has been checked is insufficient to guarantee that metadata access control is enforced correctly.

QAL implements constraint rules by defining rewriting rules for queries and updates. Combined with formal semantics, QAL enables stating properties of the system in formal logic and writing proofs that such properties are maintained by the rules.

In this section we show two example of using QAL to address some of the challenges in Section 1, and showcase how the generality of QAS allows us to rapidly implement solutions to these challenges. We look at metadata access control and metadata indexing.

Baseline System

The Setup is

- Unmodified iRODS 4.2 database in Postgres
- A mapping generated from the iRODS schema definition.

The baseline system imports relations from a source for use within rewrite rules. Example rewrite rules are shown for query, insert and delete.

```
import all from ICAT
export META
```

```
rewrite META(x, m)
  OBJT_METAMAP_OBJ(x, m)

rewrite insert META(x, m)
  insert OBJT_METAMAP_OBJ(x, m)

rewrite delete META(x, m)
  delete OBJT_METAMAP_OBJ(x, m)
```

Metadata Access Control

- Neo4j database schema for storing metadata access control information. Here x is an object id, m is a metadata attribute, user is a user id, and acc is an access control.

```
META_ACCESS_OBJ(x, m, user, acc).
```

- Make accessible a new predicate that has metadata access control

```
import all from ICAT
import META_ACCESS_OBJ from Neo4j
export META
```

Given the client_user_name and client_zone, return the user_id.

```
rewrite CLIENT_ID(u)
  USER_NAME(u, client_user_name) USER_ZONE_NAME(u, client_zone)
```

Get the user_id of the current client, then check whether has access to the metadata.

```
rewrite META(x, m)
  CLIENT_ID(user)
  OBJT_METAMAP_OBJ(x, m)
  META_ACCESS_OBJ(x, m, user, acc) Neo4j.eq(acc, 1200)
```

Insert metadata if the current client has access to the object and set access to the metadata for the current client.

```
rewrite insert META(x, m)
  CLIENT_ID(user)
  OBJT_ACCESS_OBJ(x, user, acc) eq(acc, 1200)
  insert OBJT_METAMAP_OBJ(x, m) META_ACCESS_OBJ(m, x, user, 1200)
```

If the current client has access to the metadata, delete the metadata and access to the metadata for all users.

```
rewrite delete META(x, m)
  CLIENT_ID(user)
  META_ACCESS_OBJ(m, x, user, acc) Neo4j.eq(acc, 1200)
  (delete OBJT_METAMAP_OBJ(x, m) |
    META_ACCESS_OBJ(m, x, user2, acc2) delete META_ACCESS_OBJ(m, x,
user2, acc2))
```

Metadata Indexing

- ElasticSearch for storing metadata matching regex `searchable.*`
- Make accessible a new predicate that has metadata indexing

```
Query ElasticSearch and query the iCAT catalog and combine the
results.

rewrite META_2(x, m)
  ES_META.OBJT_METAMAP_OBJ(x, m) | OBJT_METAMAP_OBJ(x, m)
```

Apply a regular expression match on the metadata. If it matches insert into ElasticSearch, otherwise insert into iCAT catalog.

```
rewrite insert META_2(x, m)
  ( ~ RegexDB.like_regex(m, "searchable.*") insert
OBJT_METAMAP_OBJ(x, m) |
    RegexDB.like_regex(m, "searchable.*") insert
ES_META.OBJT_METAMAP_OBJ(x, m) )
```

Formalization

An ongoing task is creating a formally defined semantics for QAL. This enables us to provide three types of guarantees:

- Ensure semantic consistency of translated queries of different databases. This ensures representation independence.
- Transform queries to semantically equivalent queries and optimize their performance in future work. This allows us to optimize a query in a representation independent manner.
- Explore proving correctness of policies defined in QAL. This allows us to prove that the system is configured correctly for the domain application in a representation independent manner.

The key challenge of such a formalization is to provide a way to map multiple semantics into a generic query. For example, consider a query that retrieves data from two different types of databases and combines the results. The QAL query must be divided into two parts, with each part translated to one type of database. To give semantics to this query means that we must simultaneous map semantics of both types of databases to the same generic query. To solve this problem, both the

syntax and semantics of a QAL query is defined in a parametric manner. The semantic function which maps a QAL term to an object in the domain of QAL semantics then is also parametric. Currently, we have defined this general semantic function and showed that the semantics form a near-semiring. This gives us preliminary algebraic laws to transform QAL queries for optimization. We have also defined semantics of a subset of SQL based on tables and unnamed columns. This allows us to define a translation of QAL to SQL. The next step is to prove that this translation preserves semantics and extend this approach to other query languages.

Summary

This paper described QueryArrow, generic middleware software that provides a semantically unified query and update interface to a wide range of metadata sources.