QueryArrow: Bidirectional Integration of Multiple Metadata Sources

Hao Xu$^1$

Ben Keller$^2$  Antoine de Torcy$^2$  Jason Coposky$^2$

$^1$DICE

$^2$iRODS Consortium

iRODS User Group Meeting, 2016
Table of Contents

1 Introduction

2 QueryArrow Service

3 QueryArrow Language

4 Examples
   - Metadata Access Control
   - Metadata Indexing

5 Formalization
Acknowledgement

This research is partially supported by DataNet Federation Consortium and iRODS Consortium
About me

- work at DICE on iRODS since 2010
- main developer/designer of iRODS rule engine since the 3.0 release
- main designer/initial implementation of pluggable rule engine architecture (merged into 4.2 and further developed by iRODS consortium)
  


  both papers can be found at http://irods.org/documentation/articles/

- main designer/implementor of QueryArrow (GenQuery Version 2)
Motivating Applications

Big Metadata Challenge:

- Aggregation: integrating metadata from multiple metadata sources
- Policies:
  - Procedures (do X): for example auditing
  - Constraints (ensure X): access control, fine-grained, configurable
- Discovery: metadata based indexing
- Migration: decoupling of metadata storage from metadata use, reducing downtime
Motivating Applications

Big Metadata Challenge:

- Aggregation: integrating metadata from multiple metadata sources
- Policies:
  - Procedures (do X): for example auditing
  - Constraints (ensure X): access control, fine-grained, configurable
- Discovery: metadata based indexing
- Migration: decoupling of metadata storage from metadata use, reducing downtime
Motivating Applications

Big Metadata Challenge:

- Aggregation: integrating metadata from multiple metadata sources
- Policies:
  - Procedures (do X): for example auditing
  - Constraints (ensure X): access control, fine-grained, configurable
- Discovery: metadata based indexing
- Migration: decoupling of metadata storage from metadata use, reducing downtime
Motivating Applications

Big Metadata Challenge:

- Aggregation: integrating metadata from multiple metadata sources
- Policies:
  - Procedures (do X): for example auditing
  - Constraints (ensure X): access control, fine-grained, configurable
- Discovery: metadata based indexing
- Migration: decoupling of metadata storage from metadata use, reducing downtime
Limitations of the Current Design

- No support for NoSQL databases
- One-directional w.r.t. query/update
- Partially schema dependent
- Lack of fine-grained control of execution order
- No policy support
- No formal specification
Limitations of the Current Design

- No support for NoSQL databases
- One-directional w.r.t. query/update
- Partially schema dependent
- Lack of fine-grained control of execution order
- No policy support
- No formal specification

One instance
Limitations of the Current Design

- One instance
- No support for NoSQL databases
Limitations of the Current Design

- One instance
- No support for NoSQL databases
- One-directional w.r.t. query/update
Limitations of the Current Design

- One instance
- No support for NoSQL databases
- One-directional w.r.t. query/update
- Partially schema dependent
Limitations of the Current Design

- One instance
- No support for NoSQL databases
- One-directional w.r.t. query/update
- Partially schema dependent
- Lack of fine-grained control of execution order
Limitations of the Current Design

- One instance
- No support for NoSQL databases
- One-directional w.r.t. query/update
- Partially schema dependent
- Lack of fine-grained control of execution order
- No policy support
Limitations of the Current Design

- One instance
- No support for NoSQL databases
- One-directional w.r.t. query/update
- Partially schema dependent
- Lack of fine-grained control of execution order
- No policy support
- No formal specification

Diagram:

- iRODS Core
  - GenQuery, Specific Query
  - API
  - Postgres Database Plugin
    - SQL
      - Postgres

mapping
Requirements of QueryArrow

- **Generic**
  - Representation Independent
  - Configurable

- **Formal**
  - Formalization
  - Formally Provable
    - Algorithm vs. Code
    - Paper vs. Machine Checkable
Table of Contents

1. Introduction

2. QueryArrow Service

3. QueryArrow Language

4. Examples
   - Metadata Access Control
   - Metadata Indexing

5. Formalization
Solution

- irodsAgent
- iRODS Core
- GenQuery, Specific Query
- API
- QAS Database Plugin
- ØMQ
- QAL
- QueryArrow
- Postgres QAP
  - SQL
  - Postgres
- Neo4j QAP
  - Cypher
  - Neo4j
- ElasticSearch QAP
  - EQL
  - ElasticSearch

- QueryArrow Service (QAS): Register Databases and Execution of QAL
- QueryArrow Language (QAL): Configuration, QL/DML
- QueryArrow Plugins (QAP): Mappings between QAL and Databases
For succinctness, we consider a simplified form of metadata, which is a just a tag on a data object. In our mapping this is represented by
\text{OBJT\_METAMAP\_OBJ}(x, m).
Execution of QAL across Multiple Databases

Namespace

- Postgres QAP
- Neo4j QAP
- ElasticSearch QAP

QAS

ICAT

Neo4j

ES_META
Execution of QAL across Multiple Databases

Predicates
Return all metadata associated with data objects named nx1

\[
\text{ICAT}.\text{DATA\_NAME}(x, "nx1")
\]
\[
\text{(Neo4j\_OBJT\_METAMAP\_OBJ}(x, m) \mid
\text{ES\_META\_OBJT\_METAMAP\_OBJ}(x, m))
\]

return m

User submits query. Note: Use export to avoid explicitly specifying namespace.
Execution of QAL across Multiple Databases

Parse query

- ICAT.DATAR_NAME(x,"nx1")
- Neo4j.OBJT_METAMAP_OBJ(x,m)
- ES_META.OBJT_METAMAP_OBJ(x,m)

QAS

Postgres QAP

Neo4j QAP

ElasticSearch QAP
Execution of QAL across Multiple Databases

ICAT.DATA_NAME(x,"nx1")

⊙

Neo4j.OBJT_METAMAP_OBJ(x,m)  ES_META.OBJT_METAMAP_OBJ(x,m)

⊗

QAS

DATA_NAME(x,"nx1")

Postgres QAP  Neo4j QAP  ElasticSearch QAP

retrieve data id from ICAT
Execution of QAL across Multiple Databases

ICAT.DATA_NAME(x,"nx1")

⊕

Neo4j.OBJT_METAMAP_OBJ(x,m)  ES_META.OBJT_METAMAP_OBJ(x,m)

QAS

x=[10000]

retrieve data id from ICAT

Postgres QAP  Neo4j QAP  ElasticSearch QAP
Execution of QAL across Multiple Databases

- ICAT.DATANAME(x,"nx1")
- Neo4j.OBJT_METAMAP_OBJ(x,m)  ES_META.OBJT_METAMAP_OBJ(x,m)

QAS

- OBJT_METAMAP_OBJ(10000,m)

Postgres QAP  Neo4j QAP  ElasticSearch QAP

retrieve metadata from Neo4j
Execution of QAL across Multiple Databases

```
ICAT.DATA_NAME(x,"nx1")
```


```
Neo4j.OBJT_METAMAP_OBJ(x,m) ES_META.OBJT_METAMAP_OBJ(x,m)
```

```
m=["tag1","tag2"]
```

```
QAS
```

```
Postgres QAP Neo4j QAP ElasticSearch QAP
```

retrieve metadata from Neo4j
Execution of QAL across Multiple Databases

ICAT.DATA_NAME(x,"nx1")

Neo4j.OBJT_METAMAP_OBJ(x,m) ES_META.OBJT_METAMAP_OBJ(x,m)

QAS

Postgres QAP

Neo4j QAP

ElasticSearch QAP

retrieve metadata from ES_META
Execution of QAL across Multiple Databases

\[
\times \quad \text{ICAT.DATA_NAME}(x,"nx1") \\
\oplus \quad \text{Neo4j.OBJT_METAMAP_OBJ}(x,m) \quad \text{ES_META.OBJT_METAMAP_OBJ}(x,m)
\]

\[
\text{QAS} \\
\text{Postgres QAP} \quad \text{Neo4j QAP} \quad \text{ElasticSearch QAP}
\]

\[m=["tag1'","tag2'"]\]

retrieve metadata from ES_META
Execution of QAL across Multiple Databases

m=["tag1","tag2","tag1’","tag2’"]

QueryArrow

Postgres QAP

Neo4j QAP

ElasticSearch QAP

return m=["tag1","tag2","tag1’","tag2’"]
Supported Databases by QAS

- Relational DB: Postgres, Sqlite, CockroachDB (Distributed Key-Value Store)
- Graph DB: Neo4j
- Document DB: ElasticSearch (REST API based metadata source)
- InMemory: EqDB, MapDB, MutableMapDB, RegexDB
Features Supported across All Supported Databases by QAS

- Query multiple database paradigms and aggregate results
- Dispatch update to multiple database paradigms
- Distributed transaction on databases that support two-phase commit
- Namespacing
- Regular expression, disjunctive query, multiple metadata item, ...
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>QueryArrow Service</td>
</tr>
<tr>
<td>3</td>
<td>QueryArrow Language</td>
</tr>
<tr>
<td>4</td>
<td>Examples</td>
</tr>
<tr>
<td></td>
<td>Metadata Access Control</td>
</tr>
<tr>
<td></td>
<td>Metadata Indexing</td>
</tr>
<tr>
<td>5</td>
<td>Formalization</td>
</tr>
</tbody>
</table>
Why Another Language?

- SQL is strong in query but weak in data manipulation.
- SQL performance is dependent on individual DB’s query optimizer. Need to craft different SQL for different DB to achieve optimal performance.
- SQL doesn’t support the notation of multiple distributed databases.
- SQL has very limited, unidirectional support for transforming queries (needed for policy).
- SQL cannot be easily translated to other database paradigms.
QueryArrow Language is based on/shares similar features with:

- Relational Algebra
- Process Algebra (Nearsemiring)
- Substructural Logic
- Term Rewriting
- Prolog (Datalog), iRODS 2.x rule language
- SRB query language
The QueryArrow Language

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

t ::= $i | s | v$

a ::= $N.P(t_1, \ldots, t_n) | P(t_1, \ldots, t_n)$

c ::= $a | \text{insert } a | \text{delete } a | \text{transactional}$

$\sim c | \text{exists } v.c | \text{one} | \text{zero} | c \land c | c | c$

$\text{return } v_1 \ldots v_n$

R ::= $\text{rewrite } a \land c | \text{rewrite } \text{insert } a \land c$

$| \text{rewrite } \text{delete } a \land c$

I ::= $\text{import qualified? } (all | P_1, \ldots, P_n) \text{ from } N$

E ::= $\text{export qualified? } (all | P_1, \ldots, P_n) (\text{from } N)?$
Examples of the QueryArrow Language

- Return all data objects ids and their names
  \[
  \text{DATA\_NAME}(x, y) \text{ return } x \ y
  \]

- Return all data objects names in collection \( c \)
  \[
  \text{COLL\_NAME}(x, "c") \ \text{DATA\_COLL\_ID}(y, x) \ \text{DATA\_NAME}(y, z) \ \text{return } z
  \]

- Return all data objects names in collection \( c \) or \( c2 \)
  \[
  (\text{COLL\_NAME}(x, "c") \ \text{OR} \ \text{COLL\_NAME}(x, "c2"))
  \ \text{DATA\_COLL\_ID}(y, x) \ \text{DATA\_NAME}(y, z) \ \text{return } z
  \]

- Return all data objects that do not belong to collection \( c \)
  \[
  \text{DATA\_COLL\_ID}(y, x) \ \text{DATA\_NAME}(y, z) \ \text{NOT}\ \text{COLL\_NAME}(x, "c")
  \ \text{return } z
  \]

- Insert a new data object named \( a \)
  \[
  \text{nextid}(x) \ \text{insert} \ \text{DATA\_OBJ}(x) \ \text{DATA\_NAME}(x, "a")
  \]

- Delete all data objects named \( a \)
  \[
  \text{DATA\_NAME}(x,"a") \ \text{delete} \ \text{DATA\_OBJ}(x)
  \]
Examples of the QueryArrow Language

- Return all data objects ids and their names
  \[\text{DATA\_NAME}(x, y) \text{ return } x \ y\]

- Return all data objects names in collection \(c\)
  \[\text{COLL\_NAME}(x, "c") \text{ DATA\_COLL\_ID}(y, x) \text{ DATA\_NAME}(y, z) \text{ return } z\]

- Return all data objects names in collection \(c\) or \(c2\)
  \[(\text{COLL\_NAME}(x, "c") \text{ | COLL\_NAME}(x, "c2"))\]
  \[\text{DATA\_COLL\_ID}(y, x) \text{ DATA\_NAME}(y, z) \text{ return } z\]

- Return all data objects that do not belong to collection \(c\)
  \[\text{DATA\_COLL\_ID}(y, x) \text{ DATA\_NAME}(y, z) \text{ ~COLL\_NAME}(x, "c") \text{ return } z\]

- Insert a new data object named \(a\)
  \[\text{nextid}(x) \text{ insert DATA\_OBJ}(x) \text{ DATA\_NAME}(x, "a")\]

- Delete all data objects named \(a\)
  \[\text{DATA\_NAME}(x,"a") \text{ delete DATA\_OBJ}(x)\]
Examples of the QueryArrow Language

- Return all data objects ids and their names
  \[ \text{DATA\_NAME}(x, y) \text{ return } x \text{ } y \]

- Return all data objects names in collection \(c\)
  \[ \text{COLL\_NAME}(x, "c") \text{ DATA\_COLL\_ID}(y, x) \text{ DATA\_NAME}(y, z) \text{ return } z \]

- Return all data objects names in collection \(c\) or \(c2\)
  \[ (\text{COLL\_NAME}(x, "c") \mid \text{COLL\_NAME}(x, "c2")) \]
  \[ \text{DATA\_COLL\_ID}(y, x) \text{ DATA\_NAME}(y, z) \text{ return } z \]

- Return all data objects that do not belong to collection \(c\)
  \[ \text{DATA\_COLL\_ID}(y, x) \text{ DATA\_NAME}(y, z) \sim \text{COLL\_NAME}(x, "c") \]
  \[ \text{return } z \]

- Insert a new data object named \(a\)
  \[ \text{nextid}(x) \text{ insert } \text{DATA\_OBJ}(x) \text{ DATA\_NAME}(x, "a") \]

- Delete all data objects named \(a\)
  \[ \text{DATA\_NAME}(x,"a") \text{ delete } \text{DATA\_OBJ}(x) \]
Examples of the QueryArrow Language

- Return all data objects ids and their names
  \[ \text{DATA\_NAME}(x, y) \text{ return } x \ y \]

- Return all data objects names in collection \( c \)
  \[ \text{COLL\_NAME}(x, "c") \text{ DATA\_COLL\_ID}(y, x) \text{ DATA\_NAME}(y, z) \text{ return } z \]

- Return all data objects names in collection \( c \) or \( c_2 \)
  \[ (\text{COLL\_NAME}(x, "c") \mid \text{COLL\_NAME}(x, "c_2")) \text{ DATA\_COLL\_ID}(y, x) \text{ DATA\_NAME}(y, z) \text{ return } z \]

- Return all data objects that do not belong to collection \( c \)
  \[ \text{DATA\_COLL\_ID}(y, x) \text{ DATA\_NAME}(y, z) \sim \text{COLL\_NAME}(x, "c") \text{ return } z \]

- Insert a new data object named \( a \)
  \[ \text{nextid}(x) \text{ insert } \text{DATA\_OBJ}(x) \text{ DATA\_NAME}(x, "a") \]

- Delete all data objects named \( a \)
  \[ \text{DATA\_NAME}(x, "a") \text{ delete } \text{DATA\_OBJ}(x) \]
Examples of the QueryArrow Language

- Return all data objects ids and their names
  \[
  \text{DATA\_NAME}(x, y) \text{ return } x \ y
  \]

- Return all data objects names in collection c
  \[
  \text{COLL\_NAME}(x, "c") \ \text{DATA\_COLL\_ID}(y, x) \ \text{DATA\_NAME}(y, z)
  \text{ return } z
  \]

- Return all data objects names in collection c or c2
  \[
  (\text{COLL\_NAME}(x, "c") \ \text{OR} \ \text{COLL\_NAME}(x, "c2"))
  \ \text{DATA\_COLL\_ID}(y, x) \ \text{DATA\_NAME}(y, z) \text{ return } z
  \]

- Return all data objects that do not belong to collection c
  \[
  \text{DATA\_COLL\_ID}(y, x) \ \text{DATA\_NAME}(y, z) \ \text{NOT} \ \text{COLL\_NAME}(x, "c")
  \text{ return } z
  \]

- insert a new data object named a
  \[
  \text{nextid}(x) \ \text{insert} \ \text{DATA\_OBJ}(x) \ \text{DATA\_NAME}(x, "a")
  \]

- delete all data objects named a
  \[
  \text{DATA\_NAME}(x,"a") \ \text{delete} \ \text{DATA\_OBJ}(x)
  \]
Examples of the QueryArrow Language

- Return all data objects ids and their names
  \[
  \text{DATA\_NAME}(x, y) \text{ return } x \ y
  \]

- Return all data objects names in collection \(c\)
  \[
  \text{COLL\_NAME}(x, "c") \ \text{DATA\_COLL\_ID}(y, x) \ \text{DATA\_NAME}(y, z) \ \text{return } z
  \]

- Return all data objects names in collection \(c\) or \(c2\)
  \[
  (\text{COLL\_NAME}(x, "c") \ \text{OR} \ \text{COLL\_NAME}(x, "c2")) \\
  \text{DATA\_COLL\_ID}(y, x) \ \text{DATA\_NAME}(y, z) \ \text{return } z
  \]

- Return all data objects that do not belong to collection \(c\)
  \[
  \text{DATA\_COLL\_ID}(y, x) \ \text{DATA\_NAME}(y, z) \ \text{NOT} \ \text{COLL\_NAME}(x, "c") \\
  \text{return } z
  \]

- Insert a new data object named \(a\)
  \[
  \text{nextid}(x) \ \text{INSERT} \ \text{DATA\_OBJ}(x) \ \text{DATA\_NAME}(x, "a")
  \]

- Delete all data objects named \(a\)
  \[
  \text{DATA\_NAME}(x, "a") \ \text{DELETE} \ \text{DATA\_OBJ}(x)
  \]
Translation Examples of the QueryArrow Language

Return all data objects ids and their names

\texttt{DATA\_NAME}(x, y) \texttt{return} x y

- **SQL**
  
  \texttt{select data\_id, data\_name from r\_data\_main}

- **Cypher**
  
  \texttt{match (var0:DataObject)}
  
  \texttt{return var0.data\_id, var0.data\_name}

- **ElasticSearch**
  
  \texttt{
  
  "query":{
    "bool":{"must":[{"term":{"obj\_type":"DataObject"}}]}\n  }\n  
  }
Translation Examples of the QueryArrow Language

Return all data objects ids and their names

DATA_NAME(x, y) return x y

- **SQL**

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

- **Cypher**

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

- **ElasticSearch**

  ```
  {
    "query":{
      "bool":{
        "must":[{"term":{"obj_type":"DataObject"}}]
      }
    }
  }
  ```
Translation Examples of the QueryArrow Language

Return all data objects ids and their names

\(
\text{DATA\_NAME}(x, y) \text{ return } x \ y
\)

- **SQL**
  ```sql
  select data_id, data_name from r_data_main
  ```

- **Cypher**
  ```cypher
  match (var0:DataObject)
  return var0.data_id, var0.data_name
  ```

- **ElasticSearch**
  ```json
  {
    "query":{
      "bool":"must":[{"term":{"obj_type":"DataObject"}}]
    }
  }
  ```
Table of Contents

1. Introduction
2. QueryArrow Service
3. QueryArrow Language
4. Examples
   - Metadata Access Control
   - Metadata Indexing
5. Formalization
Example Data Management Policy in QAL: Metadata access control

The Setup

- Unmodified iRODS 4.2 database in Postgres
- A mapping generated from iRODS schema definition.
- Neo4j database for storing metadata access control information
  \[ \text{META_ACCESS_OBJ}(x, m, \text{user}, \text{acc}). \]
- Make accessible a new predicate that has metadata access control
- Prevent access to baseline predicates in Postgres from users
import all from ICAT
export all from ICAT
export META
...
rewrite META(x, m)
   OBJT_METAMAP_OBJ(x, m)

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

rewrite delete META(x, m)
   transactional delete OBJT_METAMAP_OBJ(x, m)
import all from ICAT
import META_ACCESS_OBJ from Neo4j

export all from ICAT except OBJT_METAMAP_OBJ
export META
...

rewrite CLIENT_ID(u)
  USER_NAME(u, client_user_name) USER_ZONE_NAME(u, client_zone)
Example Data Management Policy in QAL: Adding Access Control (2)

... rewrite META(x, m, a, v, u)

CLIENT_ID(user) ①
OBJT_METAMAP_OBJ(x, m)
META_ACCESS_OBJ(x, m, user, acc) Neo4j.eq(acc, 1200) ②
Example Data Management Policy in QAL: Adding Access Control (3)

... rewrite insert META(x, m)
  transactional CLIENT_ID(user) ①
  OBJT_ACCESS_OBJ(x, user, acc) eq(acc, 1200) ②
  insert OBJT_METAMAP_OBJ(x, m)
    META_ACCESS_OBJ(m, x, user, 1200) ③
rewrite delete META(x, m, a, v, u)

transactional CLIENT_ID(user) ①

META_ACCESS_OBJ(m, x, user, acc) Neo4j.eq(acc, 1200) ②

(delete OBJT_METAMAP_OBJ(x, m) |

META_ACCESS_OBJ(m, x, user2, acc2)

delete META_ACCESS_OBJ(m, x, user2, acc2)) ③)
Example Data Management Policy in QAL: Metadata Indexing

The Setup
- Unmodified iRODS 4.2 database in Postgres
- A mapping generated from unmodified iRODS schema definition
- ElasticSearch for storing metadata matching regex `searchable.*`
- Make accessible a new predicate that has metadata indexing
Example Data Management Policy in QAL: Metadata Indexing (1)

... rewrite META_2(x, m)

\[
\text{ES\_META\_OBJT\_METAMAP\_OBJ}(x, m) | \text{OBJT\_METAMAP\_OBJ}(x, m)
\]
Example Data Management Policy in QAL: Metadata Indexing (2)

... 

```
rewrite insert META_2(x, m) transactional
  ( ~ RegexDB.like_regex(m, "searchable.*") ①
    insert OBJT_METAMAP_OBJ(x, m) |
    RegexDB.like_regex(m, "searchable.*")
    insert ES_META.OBJT_METAMAP_OBJ(x, m) ② )
```
Table of Contents

1. Introduction
2. QueryArrow Service
3. QueryArrow Language
4. Examples
   - Metadata Access Control
   - Metadata Indexing
5. Formalization
Formalization of The QueryArrow Language (WIP)

- Formally Specified Semantics of QAL
- Formally Specified Translation
  - QAL to SQL
  - QAL to Cypher
  - QAL to ElasticSearch Query Language
Functional Correctness

iRODS Core

API

Postgres Plugin

SQL

Postgres

iRODS Core

API

Neo4j Plugin

Cypher

Neo4j

QueryArrow

Xu ( DICE, iRODS Consortium)
Functional Correctness: State

iRODS Core

Plugin

API

S1

update

S2

update

query

...
Functional Correctness: Observational Equivalence

Plugin

S1    S2    ...

Plugin’

S1’    S2’    ...

Xu (DICE, iRODS Consortium)
Functional Correctness: Observational Equivalence

Plugin

S1

S1'

Plugin'

S2

S2'

...

Xu (DICE, iRODS Consortium)
Functional Correctness: Observational Equivalence

Diagram:

- **Plug-in**: S1, S2, ..., S_n
- **Plug-in'**: S1', S2', ..., S_n'

- Arrows indicate:
  - Update: S1 → S2, S2 → S_n
  - Query: S2' → S1'

Xu (DICE, iRODS Consortium)

QueryArrow

iRODS UGM 2016
Functional Correctness: Observational Equivalence

Plugin

S1

update

query

S2

update

...

Plugin'

S1'

update

query

S2'

update

...

Xu (DICE, iRODS Consortium)
Functional Correctness: What can be proved

QAL

executes on

Observable DB

- QAL translates to SQL
- SQL translates to RDB
- RDB implements Postgres
- Postgres implements Neo4j
- Neo4j is modeled by Cypher
- Cypher translates to Graph DB
- Graph DB implements Neo4j

Xu (DICE, iRODS Consortium)
Functional Correctness: What can be proved

QAL translates to SQL

executes on Observable DB

QAL translates to SQL

executes on Observable DB
Functional Correctness: What can be proved

QAL translates to SQL

executes on Observable DB

executes on RDB

Xu (DICE, iRODS Consortium)
Functional Correctness: What can be proved

QAL translates to SQL

QAL executes on Observable DB, which implements RDB

SQL executes on RDB
Functional Correctness: What can be proved

QAL translates to SQL

QAL executes on Observable DB

SQL executes on RDB

RDB implements Postgres

Xu (DICE, iRODS Consortium)
Functional Correctness: What can be proved

QAL translates to SQL

QAL executes on Observable DB

SQL executes on RDB

RDB is modeled by Postgres

Postgres implements Neo4j

Neo4j is modeled by Cypher

Cypher translates to RDB

RDB implements Postgres

Postgres executes on Postgres
Functional Correctness: What can be proved

QAL translates to Cypher

executes on Observable DB

implements Graph DB

is modeled by Neo4j