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

Acknowledgement

This research is paritally 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)

  Xu, Hao, Jason Coposky, Ben Keller, and Terrell Russell. "Pluggable Rule Engine Architecture." In iRODS User Group Meeting 2015, p. 29. 2015.

  Xu, Hao, et al. "A Method for the Systematic Generation of Audit Logs in a Digital Preservation Environment and Its Experimental Implementation In a Production Ready System." iPRES 2015.

  both papers can be found at

  `http://irods.org/documentation/articles/`
- main designer/implementor of QueryArrow (GenQuery Version 2)

Motivting 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

Motivting 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

Motivting Applications

Big Metadata Challenge:

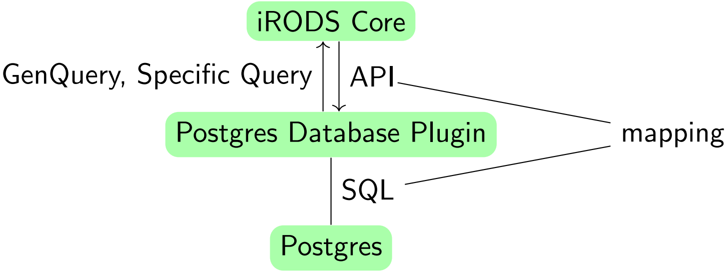
- Aggregation: integrating metadata from multiple metadata sources
- Policies:
  - Procedures (do X): for example auditing
  - Constraints (ensure X): access control, fine-grained, configurable
- Discovery: metadata based indexing
- Migration: decoupling of metadata storage from metadata use, reducing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

Motivting Applications

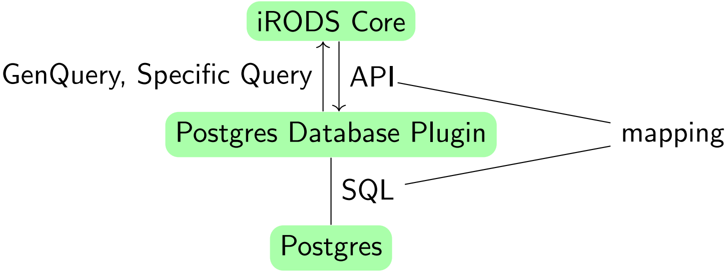
Big Metadata Challenge:

- 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

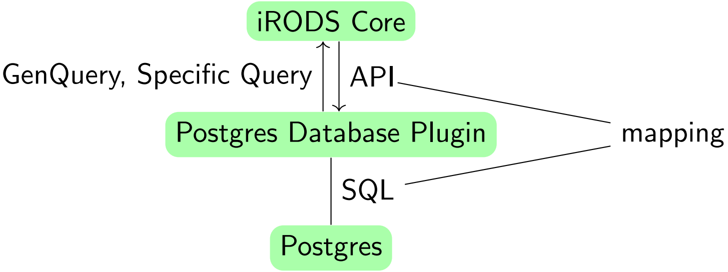


Limitations of the Current Design



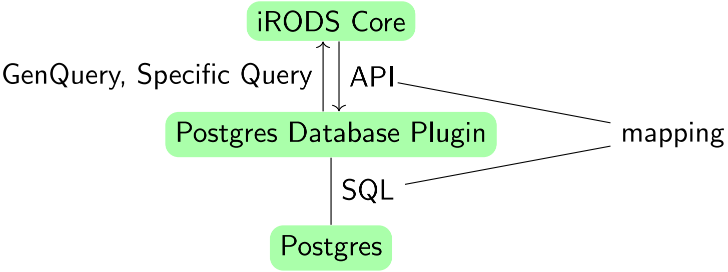
- One instance

Limitations of the Current Design



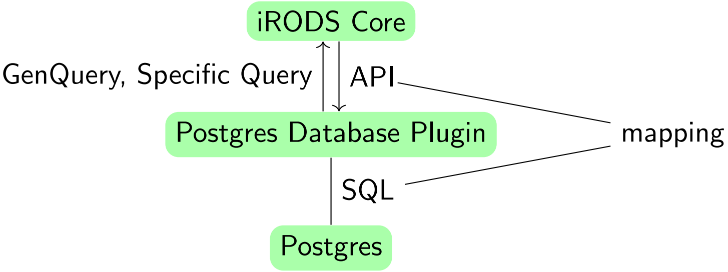
- One instance
- No support for NoSQL databases

Limitations of the Current Design



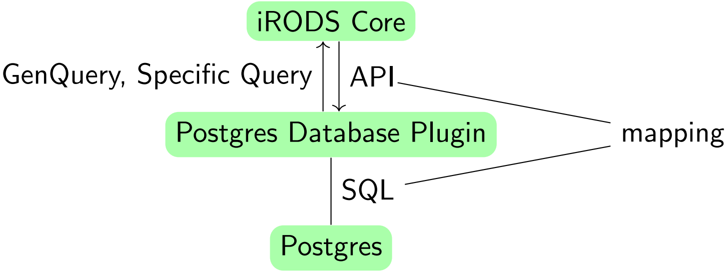
- One instance
- No support for NoSQL databases
- One-directional w.r.t. query/update

Limitations of the Current Design



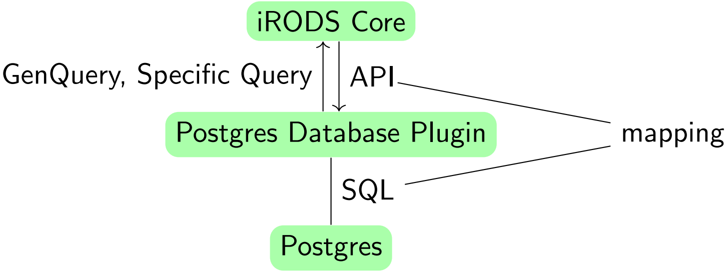
- One instance
- No support for NoSQL databases
- One-directional w.r.t. query/update
- Partially schema dependent

Limitations of the Current Design



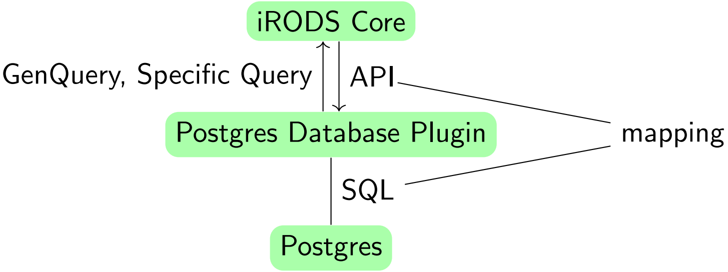
- One instance
- No support for NoSQL databases
- One-directional w.r.t. query/update
- Partially schema dependent
- Lack of fine-grained control of execution order

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



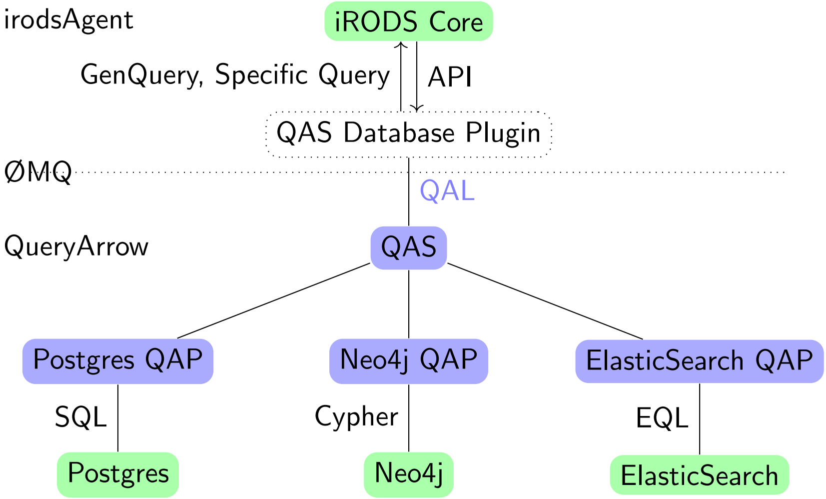
- One instance
- No support for NoSQL databases
- One-directional w.r.t. query/update
- Partially schema dependent
- Lack of fine-grained control of execution order
- No policy support
- No formal specification

Requirements of QueryArrow

- Generic
  - Representation Independent
  - Configurable
- Formal
  - Formalization
  - Formally Provable
    - Algorithm vs. Code
    - Paper vs. Machine Checkable

Table of Contents

Solution

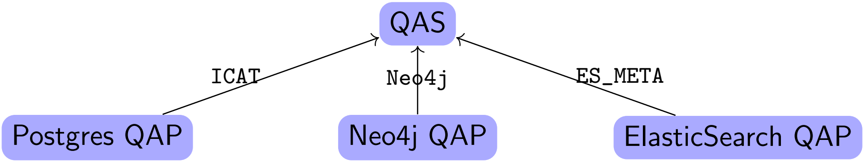


- QueryArrow Service (QAS): Register Databases and Execution of QAL
- QueryArrow Language (QAL): Configuration, QL/DML
- QueryArrow Plugins (QAP): Mappings between QAL and Databases

Execution of QAL across Multiple Databases

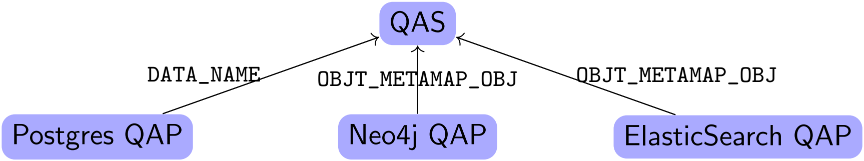
For succintness, we consider a simplified form of metadata, which is a just a tag on a data object. In our mapping this is represented by `OBJT_METAMAP_OBJ(x, m)`.

Execution of QAL across Multiple Databases



Namespace

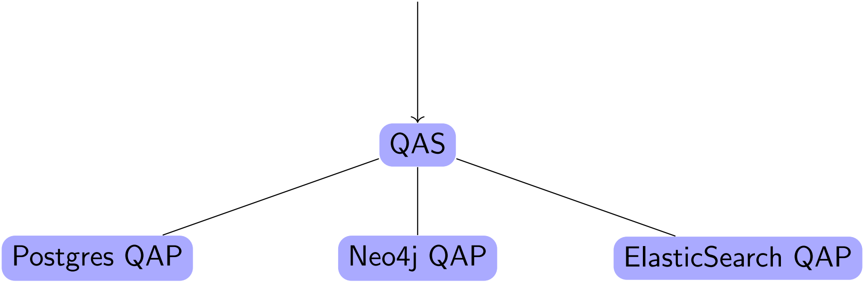
Execution of QAL across Multiple Databases



Predicates

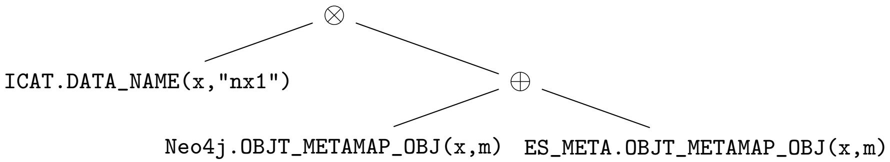
Execution of QAL across Multiple Databases

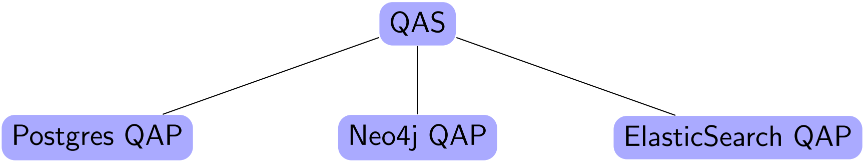
```
Return all metadata associated with data objects named nx1
ICAT.DATA_NAME(x, "nx1")
(Neo4j.OBJT_METAMAP_OBJ(x, m) |
 ES_META.OBJT_METAMAP_OBJ(x, m))
return m
```



User submits query. Note: Use export to avoid explicitly specifying namespace

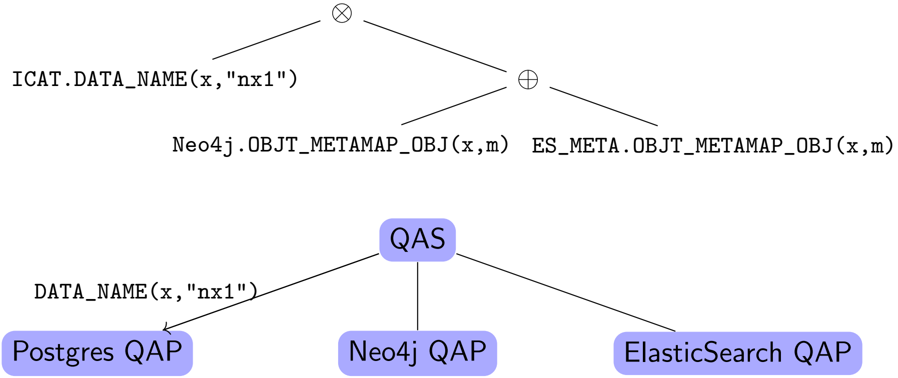
Execution of QAL across Multiple Databases





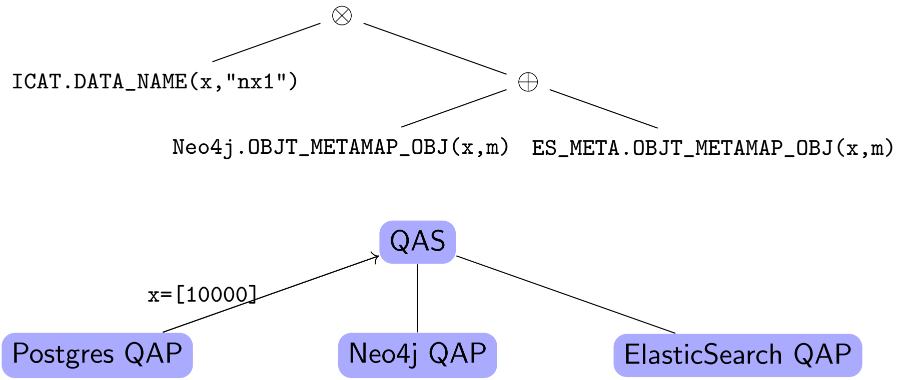
Parse query

Execution of QAL across Multiple Databases



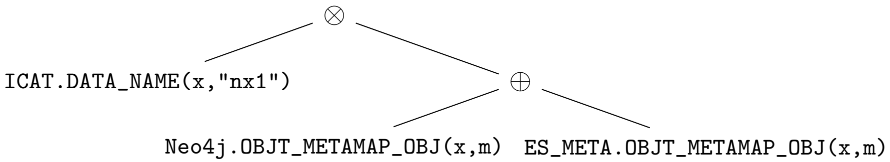
retrieve data id from ICAT

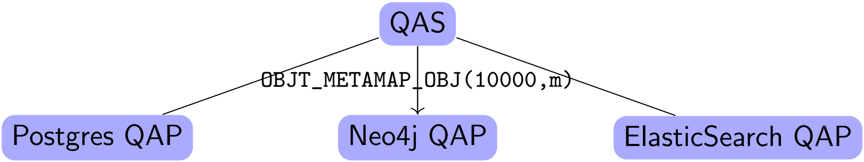
Execution of QAL across Multiple Databases



retrieve data id from `ICAT`

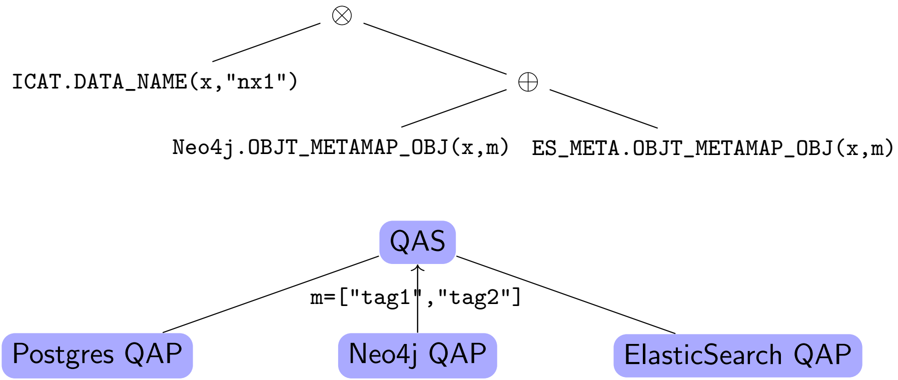
Execution of QAL across Multiple Databases





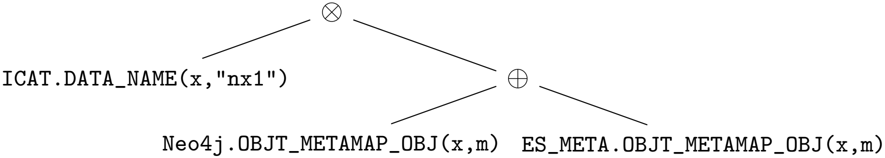
retrieve metadata from `Neo4j`

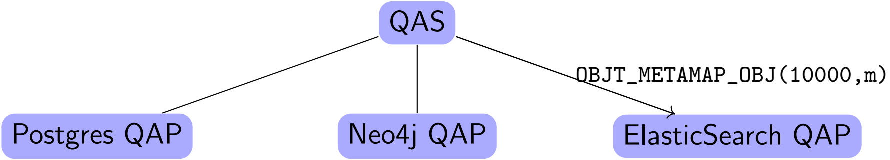
Execution of QAL across Multiple Databases



retrieve metadata from `Neo4j`

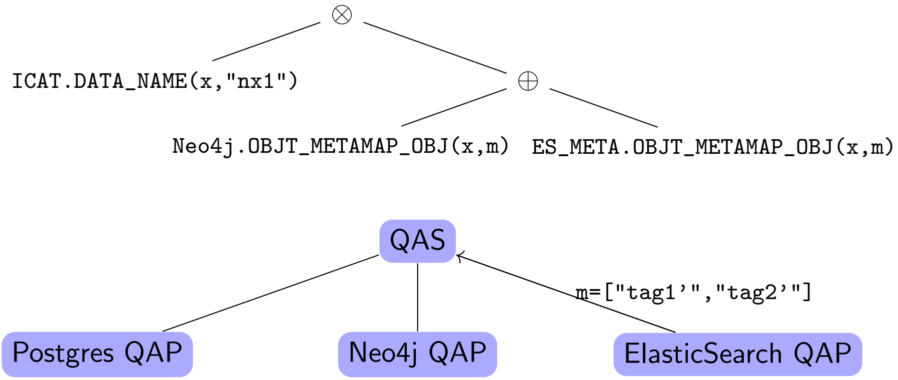
Execution of QAL across Multiple Databases





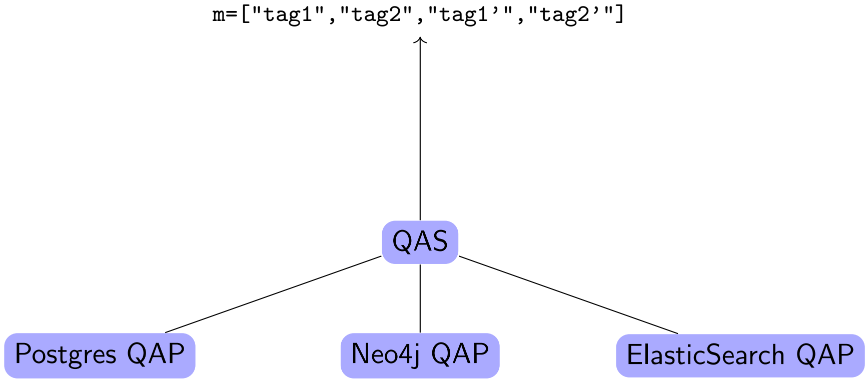
retrieve metadata from `ES_META`

Execution of QAL across Multiple Databases



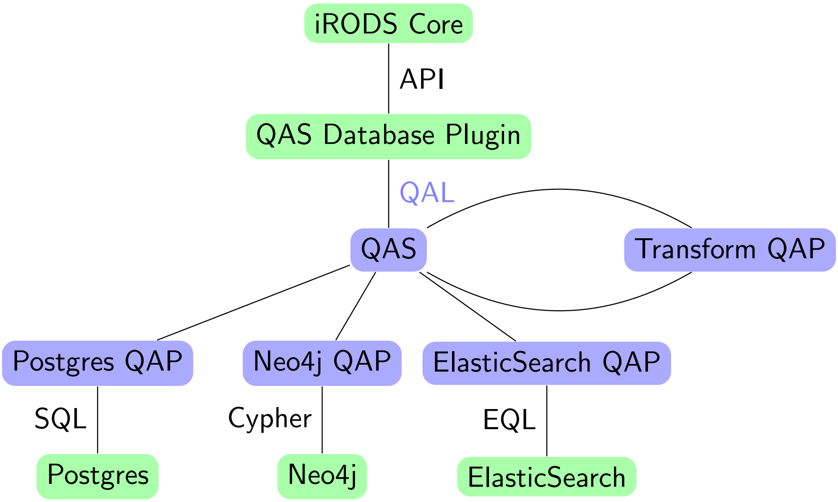
retrieve metadata from `ES_META`

Execution of QAL across Multiple Databases



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

System Diagram: Policies



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 of Contents

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

$$
\begin{array}{lcll}
t & ::= & i \mid s \mid v & \textit{terms} \\
a & ::= & N.P(t_1, \ldots, t_n) \mid P(t_1, \ldots, t_n) & \textit{atom} \\
c & ::= & a \mid \textit{insert}\ a \mid \textit{delete}\ a \mid \textit{transactional} & \\
 & & \mid \sim c \mid \textit{exists}\ v.c \mid \textit{one} \mid \textit{zero} \mid c|c \mid c\ c & \\
 & & \mid \textit{return}\ v_1 \ldots v_n & \textit{command} \\
\hline
R & ::= & \textit{rewrite}\ a\ c \mid \textit{rewrite insert}\ a\ c & \\
 & & \mid \textit{rewrite delete}\ a\ c & \textit{rewriting rules} \\
I & ::= & \textit{import qualified?}\ (\textit{all} \mid P_1, \ldots, P_n)\ \textit{from}\ N & \textit{import} \\
E & ::= & \textit{export qualified?}\ (\textit{all} \mid P_1, \ldots, P_n)\ (\textit{from}\ N)? & \textit{export}
\end{array}
$$

Examples of the QueryArrow Language

- Return all data objects ids and their names

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

- Return all data objects names in collection c

```
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

```
(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

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

- insert a new data object named a

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

- delete all data objects named a

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

Examples of the QueryArrow Language

- Return all data objects ids and their names

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

- Return all data objects names in collection `c`

```
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`

```
(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`

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

- insert a new data object named `a`

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

- delete all data objects named `a`

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

Examples of the QueryArrow Language

- Return all data objects ids and their names

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

- Return all data objects names in collection c

```
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

```
(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

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

- insert a new data object named a

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

- delete all data objects named a

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

Examples of the QueryArrow Language

- Return all data objects ids and their names

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

- Return all data objects names in collection c

```
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

```
(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

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

- insert a new data object named a

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

- delete all data objects named a

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

Examples of the QueryArrow Language

- Return all data objects ids and their names

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

- Return all data objects names in collection `c`

```
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`

```
(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`

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

- insert a new data object named `a`

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

- delete all data objects named `a`

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

Examples of the QueryArrow Language

- Return all data objects ids and their names

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

- Return all data objects names in collection `c`

```
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`

```
(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`

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

- insert a new data object named `a`

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

- delete all data objects named `a`

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

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

```
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

```
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"}}]}
 }
}
```

Table of Contents

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 `META_ACCESS_OBJ(x, m, user, acc)`.
- Make accessible a new predicate that has metadata access control
- Prevent access to baseline predicates in Postgres from users

Example Data Management Policy in QAL : Baseline

```
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)
```

Example Data Management Policy in QAL : Adding Access Control (1)

```
import all from ICAT
import META_ACCESS_OBJ from Neo4j ①
export all from ICAT except OBJT_METAMAP_OBJ ②
export META
...
rewrite CLIENT_ID(u)
  USER_NAME(u, client_user_name) USER_ZONE_NAME(u, client_zone) ③
```

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
```

Example Data Management Policy in QAL : Adding Access Control (4)

```
...
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)
  ES_META.OBJT_METAMAP_OBJ(x, m) | ①
  OBJT_METAMAP_OBJ(x, m)
```

Example Data Management Policy in QAL : Metadata Indexing (2)

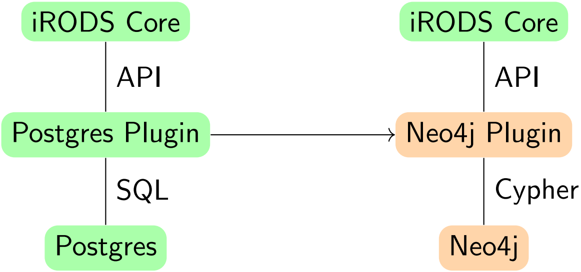
```
...

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

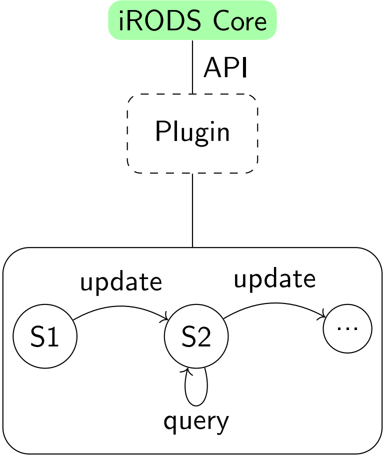
Formalization of The QueryArrow Language (WIP)

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

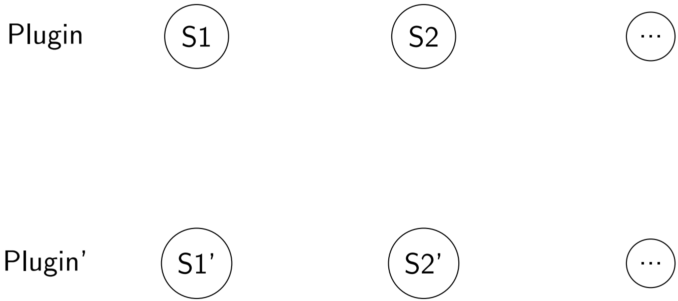
Functional Correctness



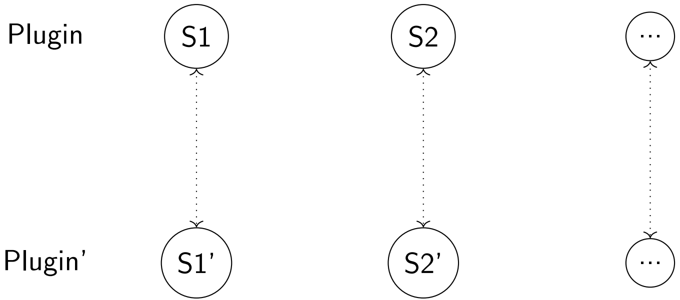
Functional Correctness : State



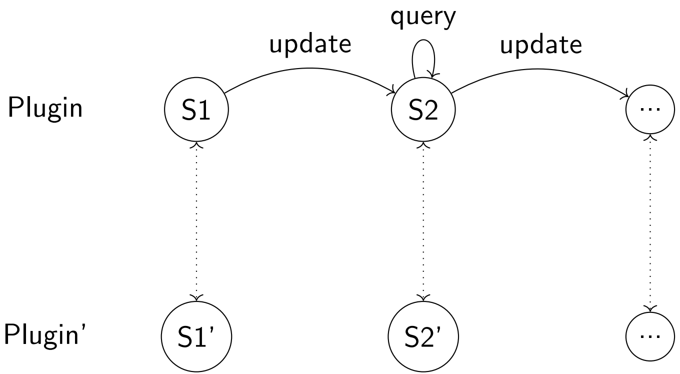
Functional Correctness : Observational Equivalence



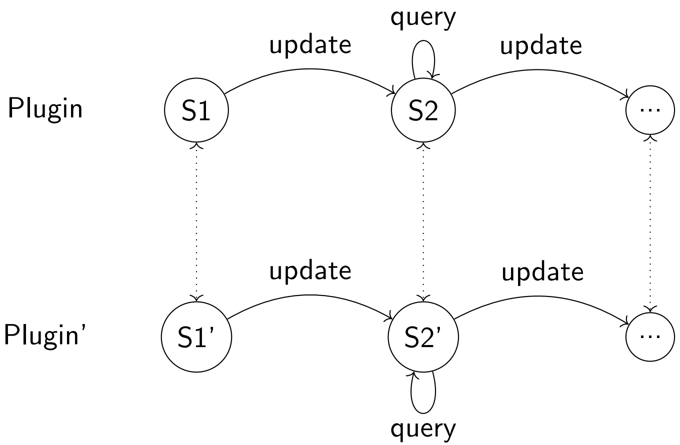
Functional Correctness : Observational Equivalence



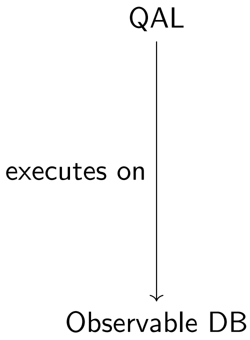
Functional Correctness : Observational Equivalence



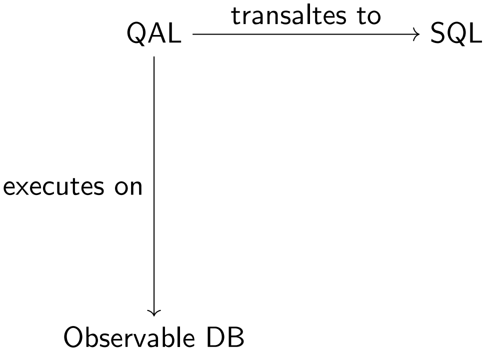
Functional Correctness : Observational Equivalence



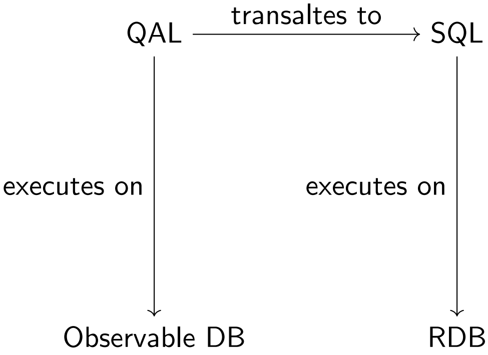
Functional Correctness : What can be proved



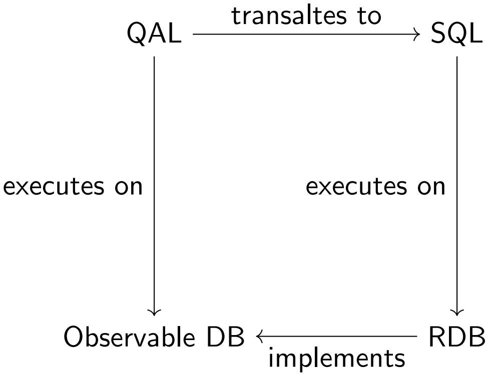
Functional Correctness : What can be proved



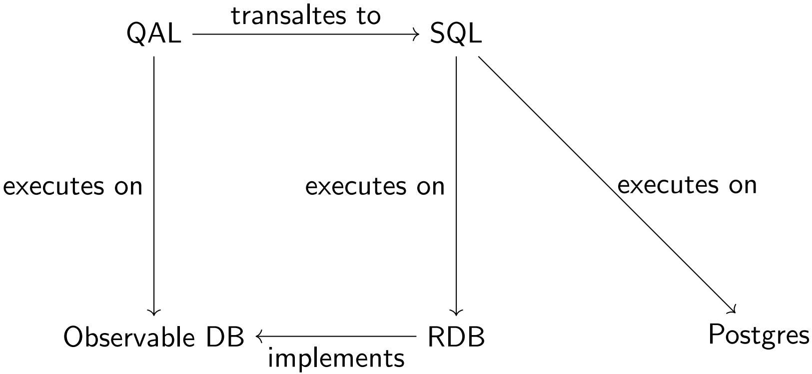
Functional Correctness : What can be proved



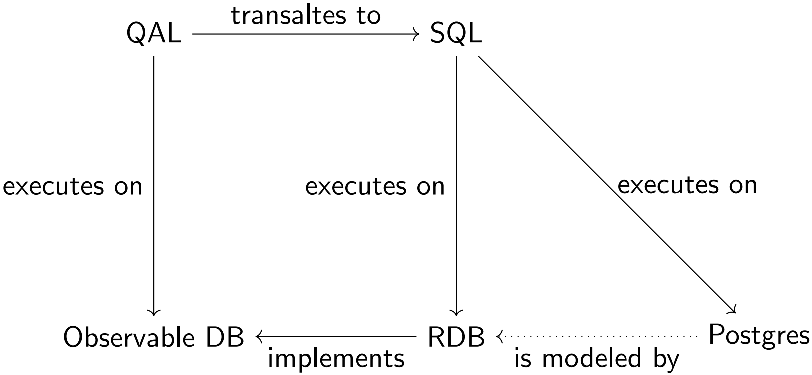
Functional Correctness : What can be proved



Functional Correctness : What can be proved



Functional Correctness : What can be proved



Functional Correctness : What can be proved