IQL Documentation

Roman Müntener, ZHAW

November 18, 2016

1 IQL

IQL (Interesting Query Language) is a domain specific query language developed for the Path Transparency Observatory (PTO) of the MAMI project.

2 Concepts

IQL is based on observations where an observation is an n-tuple consisting of at least three values. An observation is a measurement of a value for a key. A minimal complete observation is thus `(Key, Name, Value)` where `Name` is the name of the measurement such as for example *TEMPERATURE* or *HUMIDITY*. The data type of `Value` depends on `Name`. Additionally an observation may have additional attributes for example time of measurement.

3 Syntax

IQL uses JSON-encoded S-Expressions with a structure of:

```
S-EXP := {<OPERATION> : [<EXP>, <EXP>, ...]}
EXP := S-EXP | LITERAL
```

Measurement names are prefixed with a `$` sign and attributes are prefixed with a `@` sign. This is used to distinguish between strings (e.g. `'foobar'`) and the attribute *foobar* (e.g. `'@foobar'`) and the value of a measurement named *foobar* (e.g. `'$foobar'`). This is necessary because JSON only supports a few data types natively.

4 IQL Request

An IQL Request consists of `settings` and a `query` whereas `settings` is used to specify for example the amount of result sets that shall be returned or the order of results that shall be returned as well as which attribute of a tuple the requestee is interested in.

```
{"settings" : <settings>,
 "query" : <query>}
```

4.1 Settings

```
{"settings": {
     "projection" : P,
     "attribute" : A,
     "order" : [A, S]}}
```

4.1.1 Attribute

`settings.attribute` specifies the attribute that shall be selected from tuples. If set queries return sets of singletons. Some operations require `settings.attribute` to be set.

Example:

```
{"settings" : {"attribute" : "@dip"}}
```

4.1.2 Order

`settings.order` specifies the order the results shall be returned in. The first argument to order specifies the attribute to sort by and the second argument specifies the direction. `'asc'` for ascending and `'desc'` for descending.

Example:

```
{"settings" : {"order" : ['@dip', 'asc']}}
```

4.1.3 Projection

`settings.projection` allows to set a projection function that shall be applied to the selected attribute as specified in `settings.attribute`. If `settings.projection` is set, `settings.attribute` must be set as well.

Example:

```
{"settings" : {"projection" : "squ"}}
```

4.2 Query

```
{"query" : A_SET}
```

`query` contains the actual query. An aggregation operation is required.

5 Aggregation operations

5.1 all

```
{"all" : [SET]} -> A_SET
```

The `all` operation returns all tuples from its input set. It's a pseudo-aggregation.

5.2 count

5.2.1 all tuples

```
{"count": [SET]} -> A_SET
```

Returns the total count of all tuples in the input set. Result is a singleton `(count)`.

5.2.2 group members

```
{"count": [[A,...], SET]} -> A_SET
```

Groups observations based on the attributes specified in the first argument, then counts the members of each group. Result is an n-tuple with all the attibutes as specified in the first argument plus `count`.

5.2.3 group members, overwrite order

```
{"count": [[A,...], SET, S]} -> A_SET
```

Same as *group members* (see above) but overwrites `settings.order`. Third argument is either `'asc'` (ascending) or `'desc'` (descending).

5.3 sieve

```
{"sieve": [B,...]} -> A_SET
```

Behaves similar to the `sieve` set operation (see below) but it returns all paths from the sieve tree as tuples. The attribute names of the result tuples will use `:<number>` suffixes. `settings.attribute` must be set (as it is required for a sieving operation) but no final attribute selection happens meaning that `settings.attribute` has no effect on the data structure of the result set.
If you sieve over a data structure `(a,b,c)` with three steps the data structure of the result set is `(a:0,b:0,c:0,a:1,b:1,c:1,a:2,b:2,c:2)`. Ordering is done on attributes of the first step only.

6 Set operations

6.1 intersection

```
{"intersection": [SET,...]} -> SET
```

Performs a set intersection. Requires `settings.attribute` to be set. Returns set of singletons.

6.2 lookup

Returns a true set if `settings.attribute` is set unless overriden by `settings.nub`.

6.2.1 without filter

`{"lookup": [P, A, SET]} -> SET`

First argument temporarily overwrites `settings.projection` and second argument temporarily overwrites `settings.attribute`. These are overwritten for the third argument which must be a set operation. When `settings.attribute` is specified this operation returns a set of singletons as expected.
Lookup requires that the second argument is non-empty as the set expected by lookup must be a set of singletons. Lookup will return all tuples where the value of the specified attribute (with an optional projection function applied) is in the set of the third argument.

6.2.2 with filter

`{"lookup": [P, A, SET, B]} -> SET`

Behaves the same as *without filter* (see above) except that it allows for a (post-)filtering which happens after the lookup but before attribute selection (if `settings.attribute` is set, otherwise no attribute selection happens).

6.3 nub

`{"nub: " [SET]} -> SET`

Removes duplicates in the set thus converting it to a true set.

6.4 simple

`{"simple" : [B]} -> SET`

Performs a search through all tuples. First and only argument is an expression of type boolean. Returns all tuples where first expression returns true. If `settings.attribute` is set it returns a true set of singletons unless overriden by `settings.nub`.

6.5 sieve

`{"sieve": [B,...]} -> SET`

Sieve accepts a list of boolean expressions. Requires `settings.attribute` to be set. Sieving happens in steps and always returns a set of singletons.
While sieving it's possible to reference values from **previous** steps (see Example below) through the use of `:<step>` where `:<step>` is the number of the step. `{"gt": ["@time:1", "@time:0"]}` means that the value of the *time* attribute in the tuple of the second step must be larger than the one from the first step. In the very first step it searches for every tuples matching the first expression. It'll then perform a lookup and produce a tree structure. and the next expression is run on that tree structure. This step is repeated for as many expressions as were specified.
Returns a true set unless overriden by `settings.nub`.

6.5.1 Example

This example conceptually describes the process but might not accurately reflect they way sieving is implemented!

Data:

```
('L','T',15)
('L','T',16)
('L','T',20)
('Z','T',15)
('Z','T',14)
```

Structure: `(CITY,NAME,VALUE)`.

Sieve:

```
{"sieve" : [{"eq" : ["$T",15]},
            {"gt" : ["$T:1","$T:0]},
            {"gt" : ["$T:2","$T:1]}]}
```

`settings.attribute` is set to `'@CITY'`.

First step:

```
('L','T',15)
('Z','T',15)
{- no other tuples match $T = 15 -}
```

Second step:

```
('L','T',15) ('L','T',15) {- doesn't match $T:1 > $T:0 -}
             ('L','T',16) {- matches }
             ('L','T',20) {- matches }

('Z','T',15) ('Z','T',15) {- doesn't match -}
             ('Z','T',14) {- doesn't match -}
```

Third step:

```
('L','T',15) ('L','T',16) ('L','T',15) {- doesn't match $T:2 > $T:0 -}
                          ('L','T',16) {- doesn't match -}
                          ('L','T',20) {- doesn't match -}

('L','T',15) ('L','T',20) ('L','T',15) {- doesn't match -}
                          ('L','T',16) {- doesn't match -}
                          ('L','T',20) {- doesn't match -}
```

Result:

```
(L)
```

6.5.2 Performance notes

The selectivity of the first argument has performance implications. Queries will be faster the fewer matches the first argument produces.

6.6 subtraction

```
{"subtraction": [SET,...]} -> SET
```

Performs a set subtraction (set difference). Requires `settings.attribute` to be set. Returns a true set of singletons.

6.7 union

```
{"union": [SET,...]} -> SET
```

Performs a set union. If `settings.attribute` is set it returns true a set of singletons.

6.8 union-ls

```
{"union": [SET,...]} -> SET
```

Performs a set union. If `settings.attribute` is set it returns a set of singletons. `settings.nub` has no effect on this operation.

7 Expression operations

7.1 Basic operations

7.1.1 add

```
{"add": [I, I]}
{"add": [T, T]}
```

Performs addition of integers or timestamps.

7.1.2 and

```
{"and": [B,...]} -> B
```

Performs logical and.

7.1.3 div

```
{"div": [I, I]}
```

Performs integer division.

7.1.4 eq

```
{"eq": [A, A]} -> B
```

Requires both arguments to be of the same type. Returns true of arguments are equal, otherwise false.

7.1.5 ge

```
{"ge": [A, A]} -> B
```

Requires both arguments to be of the same type. Returns true if first argument is greater than or equal to the second argument, otherwise false.

7.1.6 gt

```
{"gt": [A, A]} -> B
```

Requires both arguments to be of the same type. Returns true if first argument is larger than the second argument.

7.1.7 le

```
{"le": [A, A]} -> B
```

Requires both arguments to be of the same type. Returns true if first argument is less than or equal to the second argument, otherwise false.

7.1.8 lt

```
{"lt": [A, A]} -> B
```

Requires both arguments to be of the same type. Returns true if first argument is less than the second argument.

7.1.9 mul

```
{"mul": [I, I} -> I
```

Performs integer multiplication.

7.1.10 or

```
{"or": [B,...]} -> B
```

Performs logical or.

7.1.11 sub

```
{"sub": [I, I]}
{"sub": [T, T]}
```

Performs subtraction of integers or timestamps.

7.2 Date/time operations

7.2.1 day

```
{"day": [T]} -> I
```

Returns the day (of month) of a timestamp.

7.2.2 hour

```
{"hour": [T]} -> I
```

Returns the hour part of a timestamp.

7.2.3 minute

```
{"minute": [T]} -> I
```

Returns the minute part of a timestamp.

7.2.4 month

```
{"month": [T]} -> I
```

Returns the month part of a timestamp.

7.2.5 second

```
{"second": [T]} -> I
```

Returns the second part of a timestamp.

7.2.6 year

```
{"year": [T]} -> I
```

Returns the year part of a timestamp.

8 Macros

Macros are operations operating only on constant literals.

8.1 time

```
{"time": [S]} -> T
```

Converts first argument to timestamp.

9 Restrictions

It is illegal to reference different measurement values within the same sub-expression of a set operation. Thus, the following IQL is illegal:

```
{"simple" : [{"or": [{"eq": ["$ecn.connectivity","broken"]},
                     {"eq": ["$ecn.negotiated",0]}]}]}
```

The first use of `$ecn.connectivity` binds the whole expression to it and thus the reference to `$ecn.negotiated` is illegal. However, the following IQL is legal:

```
{"union" : [{"simple": [{"eq" : ["$ecn.connectivity","broken"]}]},
            {"simple": [{"eq" : ["$ecn.negotiated", 1]}]}]}
```

10 Attribute selection

When `settings.attribute` is specified set operations will perform attribute selection which selects a single attribute (with an optional projection function applied) and thus these operations will return sets of singletons.

10.1 Example

Data:

```
('L','T',15)
('L','T',16)
```

Data structure is: `(CITY,NAME,VALUE)`.

Request:

```
{"settings": {"projection" : "squ",
              "attribute" : "$T"},
 "query": {"simple": [{"eq":[9,9]}]}}
```

IQL itself does not define any projection function except that an empty or non-present projection function must be an identity function `f(x) = x`. Projection functions are database specific and thus part of the schema. In this example we'll assume that `squ := f(x) = x * x`.

Result:

```
(225)
(256)
```

11 Examples

11.1 E1

```
{ 'query': { 'count': [ ['@dip', '$ecn.connectivity'],
                        { 'lookup': [ '',
                                      '@dip',
                                      { 'sieve': [ { 'eq': [ '$ecn.connectivity',
                                                             'works']},
                                                   { 'eq': [ '$ecn.connectivity',
                                                             'broken']}]}]}]},
 'settings': {'order': ['@dip', 'asc']}}
```

Select all dips (destination IP) where an observation with `ecn.connectivity = broken` exists and an observation with `ecn.connectivity = works` exists. Then group by `(@dip, $ecn.connectivity)` and count `$ecn.connectivity`. Example output (JSON):


```
{"count": 1, "dip": "101.200.161.203", "value": "broken"}
{"count": 1, "dip": "101.200.161.203", "value": "works"}
{"count": 1, "dip": "101.200.208.216", "value": "broken"}
{"count": 1, "dip": "101.200.208.216", "value": "works"}
{"count": 1, "dip": "101.201.104.55", "value": "broken"}
```

11.2 E2

```
{"query" : {"count" : [["@sip","$ecn.connectivity"],{"simple":[{"eq":[1,1]}]},"desc"]}}
```

Counts how many times each value for `ecn.connectivity` exists for a `sip` (start IP). Example output (JSON):


```
{"count": 616066, "sip": "139.59.249.205", "value": "works"}
{"count": 332318, "sip": "104.131.31.32", "value": "works"}
{"count": 24095, "sip": "2400:6180:0000:00d0:0000:0000:0b76:a001", "value": "works"}
```