ICARUS2 Query Language Specification

Markus Gärtner

Contents

1 Introduction

Queries in IQL are designed to be self-contained with logical sections for specifying all the information required to determine the target of a query and its granularity, resolve additional dependencies such as extensions or scripts, link and validate constraints to parts of the target

corpus or corpora and finally optional pre- and post-processing steps. To achieve this complex task IQL embeds a keyword-based syntax for the query payload within a JSON-LD structure to drive declaration of all the aforementioned information. As a side effect queries can become quite verbose and potentially cumbersome to define manually. As a countermeasure the overall structure of a query is composed of blocks that can be glued together incrementally and that make it very easy for an application built on top of it to provision boilerplate query code based on settings or a GUI so that the user only needs to type the actual constraints used in the query (the query payload). This document lists the basic building blocks of queries and their compositions.

2 Query Structure

```
{
  @context :
}
```

3 Query Preamble Section

The initial part of every IQL query is meant to introduce the version of the query language to be used and all the desired extensions or configurations. In addition it is responsible to bind all the later used identifiers for top level corpus members to unambiguous targets (either via manifests or to actual corpus members). This part of a query can become very verbose and is intended to be automatically generated from the host application based on the search context or GUI choices made by the user.

3.1 Dialect

IQL is designed as an evolving query language that can be altered and changed in terms of syntax as the need arises. To enable evaluation engines to easily perform compatibility checks or reject a query, the exact dialect used can be declared by starting the preamble with a “DIALECT xyz“ part, where “xyz“ is a version definition. The initial version of IQL

is “1.0“ and leaving the dialect part of a query blank, this initial version will be assumed!

3.2 Imports

To allow for flexible integration of macro definitions or bigger language extensions, IQL provides an optional “IMPORT“ section in the preamble that lets users specify exactly what additional modules besides the bare IQL core are required for evaluating the query. each import target is specified by declaring a URI for its resolution and an optional namespace alias to be used within the query.

“‘ importStatement := ’IMPORT’ importTarget (’,’ importTarget)* importTarget := StringLiteral (’AS’ Identifier)? “‘

3.3 Switches

For increased flexibility, IQL supports a collection of switches to turn certain optional features on when needed. Switches are part of a query’s preamble and cannot be changed after their initial declaration. All the native IQL switches use the prefix “iql“ for their name. Any extensions that offer additional switches should declare and use their own namespace for switches!

Currently supported switches:

| Name | Description | |:———————— |:—————————— | | iql.string.case.off | Turns of case sensitivity when performing string operations such as equality checks. | | iql.string.case.lower | Another approach to case insensitivity, this switch turns all strings into lower case. | | iql.expansion.off | Effectively shuts down value expansion section 12.12. | | iql.string2bool.off | Deactivates the interpretation of strings as boolean values as described in section 13.1. | | iql.int2bool.off | Deactivates the interpretation of integers as boolean values as described in section 13.1. | | iql.float2bool.off | Deactivates the interpretation of floating point numbers as boolean values as described in section 13.1. | | iql.obj2bool.off | Deactivates the interpretation of arbitrary objects as boolean values as described in section 13.1. | | iql.any2bool.off | Deactivates the interpretation of anything non-boolean as boolean value. This is a combination of “iql.string2bool.off“, “iql.int2bool.off“, “iql.float2bool.off“ and “iql.obj2bool.off“. | | iql.direction.reverse | Reverses the direction

used to traverse corpus data for a search. | | iql.array.zero | Change array access section 12.4 to be 0-based. | | iql.warnings.off | Deactivates all warnings, potentially resulting in confusing results if there are mistakes in the query. |

3.4 Properties

In addition to switches, that allow to turn certain IQL feature son or off, a more fine-grained configuration can be performed using named properties. These take the form of a classic assignment of “name = value“ and support any of the basic literals section 9.

4 Query Statement Section

The only truly obligatory part of an IQL query, this section defines what to actually extract from what corpus. It is divided into three subsections that define the corpus source section 4.1, a layer/scope selection section 4.2 for vertical filtering and the actual constraints section 13.1 for horizontal filtering of the corpus data.

4.1 Corpus Source

In the initial source statement the corpora to be used are defined and optionally re-assigned new namespaces for easier use.

“‘ sourceStatement := ’FROM’ corpusSelector (’,’ corpusSelector)*
corpusSelector := (qualifiedIdentifier | StringLiteral) (’AS’ Identifier)?
qualifiedIdentifier := Identifier (’::’ Identifier)* “‘

4.2 Layer Select

After the corpus source definition, this section is the first actual filtering step in the query, by declaring what layers, layer groups or scopes of the corpus are to be accessed and thereby vertically filtering the corpus data.

“‘ selectStatement := ’SELECT’ layerSelector (’,’ layerSelector)*
layerSelector := qualifiedIdentifier (’AS’ ’PRIMARY’? Identifier | ’AS’ ’PRIMARY’? ’SCOPE’ Identifier ’(’ scopeElementList ’)’)?

scopeElementList := qualifiedIdentifier '*'? (',' qualifiedIdentifier '*'?)* "'

Every layer selector either references an entire subgraph of the corpus member-graph directly or constructs a partial selection with the "SCOPE" keyword. When using the first approach, an [item layer](../../icarus2-model-api/src/main/java/de/ims/icarus2/model/api/layer/ItemLayer.java) is referenced and all its dependencies and associated annotation layers will be made available implicitly. This is an easy way of accessing simple corpora, but can lead to costly I/O overhead when loading vast parts of a complex corpus that aren't actually needed to evaluate the query. For a more fine-grained alternative, the "SCOPE" keyword allows to create a scope that spans an exactly specified collection of layers. If a layer in the list of scope elements is appended the "*" symbol, the entire member-subgraph for this layer will be added to the scope.

If multiple layer selectors are defined, up to one can be declared as "PRIMARY" to represent the granularity of returned items for the search.

5 Result Processing Section

There be dragons.

(Content of the result section will be added as IQL evolves)

6 Query Payload

The innermost part of every IQL query is the collection of constraints used to actually extract and match data from the corpus. It is composed in a keyword-style syntax to differentiate between the different *dialects* or constraint modes.

7 Reserved Words

The following list of keywords is reserved and any of the words may not be used as direct identifier strings in a query (they are reserved in both all lowercase and all uppercase variants):

NULL	TREE	IMPORT	END
TRUE	ALIGNED	LABEL	COUNT
FALSE	GRAPH	DEFAULT	DIALECT
FROM	OR	GROUP	PRIMARY
SETUP	AND	IN	APPEND
SELECT	HAVING	FOREACH	BY
AS	LOCAL	EVEN	FILTER
SCOPE	RANGE	ODD	ON
WITH	LIMIT	OMIT	RETURN
DISTINCT	ALL	STEP	APPENDIX
WHERE	NOT	DO	

sort entries in keywords file

In addition the following strictly lowercase words are reserved as type identifiers and may not be used otherwise:

int	double
long	string
float	boolean

8 Comments

IQL supports single-line comments, indicated by "//". All remaining content in a line after the comment indicator will be ignored when parsing and evaluating a query.

9 Literals

9.1 String Literals

IQL uses double quotes to define string literals. String literals may not contain any of the following symbols directly:

```
\n line break
\r carriage return
\f form feed
\b backspace
\t tab
```

```
\ backslash
" nested quotation mark
```

Any of those symbols listed above can be embedded into a string literal as part of an escape sequence with a preceding backslash.

Examples for valid string literals:

```
"string"
"123"
"some fancy number (123.456e-789) and emoji "
"a more complex string!"
"a\n multiline\n string..."
```

9.2 Boolean Literals

Boolean literals are limited to either all lowercase or all uppercase versions of the literals “true“ and “false“.

9.3 Integer Literals

9.3.1 Signed Integer Literals

Literals representing regular `int` (32bit) or `long` (64bit) integers consist of an optional initial sign (`+` or `-`) and the body consisting of digits or underscore (`_`) characters. Underscore characters may only appear inside the integer literal, never at the beginning or end (not counting the sign symbol).

Examples for valid (signed) integer literals:

```
1
+123
-123
1_000_000
-99_000000_0
```

9.3.2 Pure Integer Literals

Some parts of the IQL syntax only allow unsigned "pure" integers and will explicitly state this fact. In those special cases integer literals may neither contain the initial sign symbol nor intermediate underscores.

9.4 Floating Point Literals

Floating point literals are constructed by using a (signed) integer literal for the pre-decimal part, a dot '.' as delimiter and a decimal part made up by a unsigned integer literal. They represent either single-precision `float` (32bit) or double-precision `double` (64bit) values.

Examples for valid (signed) floating point literals:

```
1.0
+123.456
-123.456
1_000_000.999
-99_000000_0.000_000_001
```

While many languages offer to express floating point literals in the scientific notation with explicit exponent declaration, we do not include this in the initial draft of IQL.

10 Identifiers

Identifiers in IQL are combinations of lowercase or uppercase alphabetic `[a-zA-Z]` characters that may contain underscore symbols _ between the first and last position and may also contain digits `[0-9]` on any position except as initial symbol.

Examples for valid identifiers:

```
x
myIdentifier
x1
x321
some_random_id
random_2_4
notTheBest_____________example
```

Identifiers are limited in length by the engine to a total of 255 characters.

11 Variables and References

In IQL all top-level (i.e. not part of the tail expression in a hierarchical path) identifiers are expected to reference 'something' from the global namespace available to the query. This namespae is populated with all the globally available constants, methods and helper objects from the IQL core and any imported extensions, as well as all the corpus members defined in the scoping part of the query. Outside this global namespace any dynamically created identifiers from within a query reside in the variable namespace and are marked with a preceding `@` (e.g. `@myVariable`). They can be used the same way as any regular identifier, with the exception of allowing assignment expressions when inside script blocks. In addition any corpus members bound within a constraint section are prefixed with a `$` sign, such as `$token1`.

Type	Prefix	Example	Scope	Fixed Type	Final	Re-Assign
Reference	none	`max()`	global	X	X	
Variable	`@`	`@myVar`	limited	(X)		X
Member	`$`	`$token`	limited	X	(X)	

Special remarks:

Variables are more or less general-purpose storage objects for arbitrary values and without a fixed type. Their first assignment however hints at the implied type to be used and as such they can cause cast errors when used for situations where an incompatible type would be needed.

Member identifiers are final in the sense that they cannot be reassigned explicitly but will be implicitly for every iteration of the query on a new part of the corpus. For example, above `$token` member will point to a new token object every time the inner constraint parts of the query are evaluated. Therefore member identifiers could be viewed as a sort of loop variable.

12 Expressions

Expressions are the foundation of every query and can take any of the following forms:

12.1 Primary Expressions

Any literal (boolean, string, integer or floating point) can serve as a primary expression.

12.2 Path Expressions

For navigating hierarchically structured object graphs or namespaces, expressions can take the form of paths:

“‘ <expression> ’.’ <identifier> “‘

Examples:

```
someObejct.someProperty
some.really.long.winded.path
```

Note that for a lot of native classes of the ICARUS2 framework, IQL provides convenient path-based alternatives to method invocations. For example in the context of navigating a structure, “someStructure.getParent(someItem)“ can be replaced by “someItem.parent“ as long as “someStructure“ is unambiguous in the current context and already bound.

12.3 Method Invocation

Method invocations consist of an expression that points to the actual method (such as an identifier in the global namespace) and round brackets for the invocation with an optional argument list:

“‘ <expression> ’(’ <expression>? (’,’ <expression>)* ’)’ “‘

Examples:

```
myFunction()
myNamespace.someFunction(someArgument, anotherArgument)
min(123, 456, dynamicContent())
```

12.4 Array Access

Arrays are accessed by an expression pointing to the array itself and an index expression in square brackets indicating the position of the desired element within the array. Note that the index expression must evaluate to an integer value within `int` space. Positive values indicate the position beginning from the start of the array (with 1 being the first

position for better human readability per default, see section 3.3 for an option to switch to 0-based indices), whereas negative values allow backwards referencing of elements with `-1` pointing to the last array element and `-2` to the second to last one. For multidimensional arrays several index statements can be chained or even combined in a single comma-separated list.

“‘ <expression> ’[’ <expression> (’,’ <expression>)* ’]’ “‘

Examples:

```
myArray[1]
complexArray[1][2][3]
complexArray[1, 2][3]
complexArray[1, 2, 3]
```

Note that IQL provides convenient ways of using array access patterns to access list-like data structures and/or classes of the framework: Every ItemLookup implementation, such as Container or Structure that would traditionally access its content via “myStructure.getItemAt(someIndex)“ can be used the same as any regular array with the expression “myStructure[someIndex]“.

12.5 Annotation Access

The ICARUS2 framework models segmentation, structure and content of a corpus resource as different aspects. As such the information about any annotation attached to a given Item is stored apart from it and therefore is not easily accessible from the item alone. To simplify the usage of annotations within a query, IQL provides the following expression as syntactic sugar for accessing (multiple) annotations directly from an item:

“‘ <expression> ’’ <expression> (’,’ <expression>)* ’’ “‘

The first expression must evaluate to an item reference and the annotation pointers inside curly brackets must evaluate to strings (if only a single expression is given, it can evaluate to a list or array and be expanded, cf. section 12.12) that uniquely denote annotation layers in the current context of the query. Typically users will use string literals in double quotes to explicitly state the annotations to be accessed, but the IQL syntax allows for very flexible extraction statement. If the evaluation of those annotation pointers yields more than one string, the result

will be an array-like object containing the resolved values for each of the annotation keys in the same order as those were specified.

Examples:

```
myItem{"pos"}
myItem{"form", "pos", "lemma"}
firstSetValue(myItem{"parser1.head", "parser2.head"})
// extract values from multiple concurrent annotation layers
// and pick the first one present
```

12.6 Type Cast

Expressions in IQL are automatically cast to matching types according to the actual consumer's needs. Explicit casts can be performed by preceding an expression with one of the type keywords listed above in round brackets.

Examples:

```
(int) myValue
(long) 12345.678
(float) average(myVector)
```

12.7 Wrapping

Expression hierarchy and evaluation order follows the order the different types of expressions are listed here. To dictate another order, expressions can be wrapped into round brackets. This will cause the inner expression to be evaluated independent of potential hierarchical rules from outside.

Examples:

```
6 + 4 * 2   // multiplication is evaluated first -> result 14
(6 + 4) * 2 // addition is forced to be evaluated first -> result 20
```

12.8 Set Predicate

Also called 'containment predicate', this expression allows to check if a given value is a member of a specified set (or generally speaking 'collection'). The basic form of a set predicate looks as follows:

“‘ <expression> ’IN’ ’’ <expression> (’,’ <expression>)* ’’ “‘

The entire expression evaluates to a boolean value and will be “true“ iff the input expression (left-most one) evaluates to the same value as any of the expressions inside the curly brackets (the set definition). See the section about equality operators in section 12.10. Note that methods or collections used inside the set definition are subject to the expansion rules described in section 12.12. The primary use case for set expressions is to greatly simplify the declaration of constraints for multiple alternative target values.

Set predicates can be directly negated (apart from wrpping section 12.7 them and negating section 12.9 the entire expression) with with an exclamation mark “¡‘ or the keyword “NOT“ in front of the “IN“ keyword. If the input expression evaluates to an array-like object, the set predicate will expand its content and evaluate to “true“ if at least *one* of its elements is found to be contained in the set. The set predicate can be universally quantified with a star “*“ or the “ALL“ keyword in front of the opening curly bracket to change the overall behavior such that the result will be “true“ iff *all* of the elements are contained in the set.

The complete syntax with all options looks as follows:

“‘ <expression> (’NOT’ | ’!’)? ’IN’ (’ALL’ | ’*’)? ’’ <expression> (’,’ <expression>)* ’’ “‘

Examples:

```
someAnnotationValue IN {"NP","VP","-"}
someAnnotationValue NOT IN {"NN","DET"}
myValue IN {getLegalNames()}
fetchCharacterNamesInChapterOne() IN {getOrcishNames()}
```

12.9 Unary Operation

IQL only allows three unary operators to be used directly in front of an expression, the exclamation mark “¡‘ and the “NOT“ keyword for boolean negation and the minus sign “-“ for negating numerical expressions.

Examples:

```
!someBooleanFunction()
NOT someBooleanValue
-123
```

```
-myNumericalFunction()
```

12.10 Binary Operation

Binary operations between two expressions take the following simple form:

“‘ <expression> <operator> <expression> “‘

Binary operators follow an explicit hierarchy, listed below in the order of priority, from highest to lowest:

Operators	Explanation
`*  /  %`	multiplication, division and modulo
`+  -`	addition and subtraction
`<< >> &	^`	shift left, shift right, bitwise and, bitwise or, bitwise xor
`<  <=  >  >=`	less, less or equal, greater, greater or equal
`~  !~  #  !#`	string operators: matches (regex), matches not (regex), contains, contains not
`==  !=`	equals, equals not
`&&  AND`	logical and
`||  OR`	logical or

12.10.1 Basic Numerical Operations

Basic numerical operations follow the standard mathematical rules for priorities. While the basic numerical types (‘int‘, ‘long‘, ‘float‘ and ‘double‘) can be arbitrarily mixed inside those expressions, the type used during the expression and as result will be determined by the least restrictive type of any operand involved.

12.10.2 Bit Operations

Bitwise operations (&, | and ^) take integer expressions (or any other form of *bitset*) as inputs and generate a result of the corresponding type. If different types are used (e.g. `int` and `long`), one must be cast section 12.6 to match the other. If value expansion section 12.12 is active, any array-like data can also be used and will be subject to element-wise bit operations.

The two shift operations (‘«‘ and ‘»‘) take arbitrary integer types as left operand and an `int` value as right operand.

12.10.3 Ordered Comparisons

Comparisons are special binary operators that take two expressions of equal or compatible result type and produce a boolean value. Note that their exact semantics are type specific, e.g. when comparing strings, the operation is performed lexicographically.

12.10.4 String Operations

To account for the ubiquity of textual annotations in corpora, IQL provides a set of dedicated string operators to perform substring matching (with the *contains* operator `#` or its negated form `!#`) and regular expression matching (via `~` and `!~`). Per default IQL uses the Java regex syntax, but for the future, additional switches section 3.3 are planned to allow finer control over regex details.

Examples:

```
somePosAnnotation # "V"        // find verbal forms
somePosAnnotation !~ "NN|NS"   // alternative to the set predicate with more flexi
```

12.10.5 Equality

Equality checks follow the same basic conditions as ordered comparisons section 12.10.3, but with the following rules for comparable values “a“ and “b“:

```
a == b iff !(a<b) && !(a>b)
a != b iff a<b || a>b
```

More generally, equality between expressions in IQL is based on content equality and therefore type specific. Note that trying to check two expressions of incompatible types (such as `int` and “string“) for equality will always evaluate to “false“ and also emit a warning.

12.10.6 Logical Composition

All boolean expressions can be combined via disjunction (either double pipes || or the `OR` keyword) or conjunction (double ampersand && or the `AND` keyword), with conjunction having higher priority. While not strictly mandatory, evaluation of IQL expressions is recommended to employ optimized interpretation such that only the first operand is evaluated if possible. When the first operand of a disjunction evaluates to “true“, the entire expression is already determined, same for a conjunction’s first operand yielding “false“.

Examples:

```
a>1 && b<2
x==1 or x==3
```

12.11 Ternary Operation

A single ternary operation is supported in IQL, the popular if-then-else replacement with the following syntax:

“‘ <expression> ? <expression> : <expression> “‘

The first expression must evaluate to a boolean value and determines which of the following two alternatives will be evaluated for the total value of the expression. Note that the second and third expressions must have compatible result types.

Examples:

```
x<2 ? "text for smaller value" : "some other text"
```

12.12 Value Expansion

IQL supports expansion of arrays, lists and array-like method return values for situations where an immediate consumer supports lists of values as input. Assuming the method “randomPoint()“ returns an array of 3 integer values or a *array-like* data type (such as a point) and another method “invertPoint(int, int, int)“ takes 3 integer arguments, then the invocation of “invertPoint(randomPoint())“ is legal and the array or object from the inner method call will be automatically expanded into the separate 3 values. This is especially handy when dealing with multidimensional arrays, as regular indexing would require manual extraction of

method return values into variables to then be used in accessing the different array dimensions. With automatic expansion, a three-dimensional array could directly be accessed with aforementioned method via “array[randomPoint()]“.

13 Constraints

Simply put, constraints are expressions that evaluate to a boolean result. Apart from native boolean expressions (such as comparisons, boolean literals or boolean functions), IQL allows the following evaluations as syntactic sugar:

| Type | Condition | Value | |:————————— |:————————— |:———— | | “string“ | empty or null | “false“ | | `int` or `long` | “0“ | “false“ | | `float` or `double` | “0.0“ | “false“ | | any object | null | “false“ |

13.1 Constraints

The constraints section in IQL consists either of the sole “ALL“ keyword or of an optional bindings definition and the actual constraints themselves. A binding is a collection of member references section 11 that get declared to belong to a certain type and/or part of the corpus. The ’DISTINCT’ keyword enforces that the bound member references in this binding do **not** match the same target. Depending on the localConstraint used in the query, this might be redundant (e.g. when using the member references as identifiers for tree nodes who already are structurally distinct), but can still be used to make that fact explicit.

“‘ bindingsList := ’WITH’ binding (’AND’ binding)* binding := member (’,’ member)* ’AS’ ’DISTINCT’? qualifiedIdentifier “‘

Constraints are further divided into local constraints (signaled by the “WHERE“ keyword) and global ones (with the “HAVING“ keyword). Local constraints are obligatory and define the basic complexity of the query (flat, tree or graph). They also introduce certain limitations on what can be expressed or searched (e.g. a “flat“ local constraints declaration will not provide implicit access to tree information). However, global constraints can introduce arbitrary constraints and thereby increase the evaluation complexity, potentially without limits. Since there is no way for an evaluation engine to assess the complexity of user macros or extensions, extensive use of global constraints could in fact

lead to extremely slow searches or even create situations where an evaluation will never terminate at all.

13.1.1 Basic Constraints

Constraints can either be predicates, loop predicates, bracketed (with “(“ or “)“) constraints or boolean disjunctions (via “OR“) or conjunctions (with “AND“) of constraints.

Predicate Predicates are essentially expressions section 12 that evaluate to boolean value. See the previous sections on constraints section 13.1 for information on how non-boolean types are interpreted as boolean values and the switches section 3.3 section for ways to influence this behavior.

Loop Predicate TODO

13.1.2 Flat Constraints

Flat constraints provide no extra helpers to declare structural properties of the query. They consist of arbitrary basic constraints section 13.1.1 and typically make global constraints redundant.

13.1.3 Tree Constraints

TODO

13.1.4 Graph Constraints

content

13.1.5 Global Constraints

Global constraints con be any basic constraint section 13.1.1.