ICARUS2 Query Language Specification

Markus Gärtner

2020

Contents

1 Query Structure

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 so called *query payload*). This document lists the basic building blocks of queries and their compositions.

```
1 {
2   "@context" : "http://www.ims.uni-stuttgart.de/icarus/jsonld/iql/
      query"
3 }
```

2 JSON-LD Elements

say something about the namespace and general iql: prefixing

2.1 Binding

A binding associates a collection of member variables (3.5) with the content of a specific item layer or derived layer type.

Attributes of `iql:Binding`:

Attribute	Type	Required	Default
distinct	Boolean	no	false
edges	Boolean	no	false
target	string	yes	

`iql:distinct` Enforces that the bound member references in this binding do **not** match the same target items during evaluation. Depending on the structural constraints used in the query, this setting 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.

`iql:edges` Signals that the member labels are to be used for edges within a structure.

`iql:target` The name or alias of the layer to whose content the member variables should be bound.

Nested Elements of `iql:Binding`:

Element	Type	Required
members	array of `iql:Reference` (2.15)	yes

`iql:members` Non-empty collection of member references that are bound to the target layer's content. Every such instance of `iql:Reference` (2.15) must be unique within the surrounding `iql:Payload` (2.11).

2.2 Constraint

Constraints represent the actual content filtering of every query.

Attributes of `<Constraint>`:

Attribute	Type	Required	Default
id	string	yes	
solved	Boolean	no	false
solvedAs	Boolean	no	false

`iql:id` Identifier to uniquely identify the constraint within the entire query.

`iql:solved` Hint for the evaluation engine that this constraint has already been solved, either by a back-end implementation or as a result of (partial) query evaluation by the engine itself.

`iql:solvedAs` Specifies to what Boolean value (`true` or `false`) the constraint has been evaluated.

2.2.1 Predicate

Wraps a Boolean `iql:Expression` into an atomic constraint element that represents the smallest unit of evaluation for the top-level evaluation engine.

Extends `<Constraint>`(2.2).

Nested Elements of `iql:Predicate`:

Element	Type	Required
expression	`iql:Expression` (2.6)	yes

`iql:expression` The actual expression to be evaluated to a Boolean result. Note that typically this expression **cannot** be composed of directly nested Boolean conjunctions or disjunctions, as the engine will have parsed those into `iql:Term` (2.2.2) objects already during the first processing phase.

2.2.2 Term

A collection of constraints with a logical connective.

Extends `<Constraint>`(2.2).

Attributes of `iql:Term`:

Attribute	Type	Required	Default
operation	enum	yes	

`iql:operation` The Boolean connective to be applied to all the constraint items. Legal values are the strings “conjunction” or “disjunction”.

Nested Elements of `iql:Term`:

Element	Type	Required
items	array of `iql:Constraint` (2.2)	yes

`iql:items` The constraints which are to be combined by the specified `iql:operation`.

2.3 Corpus

Top-level entry point for querying a single stream.

Attributes of `iql:Corpus`:

Attribute	Type	Required	Default
id	string	yes	
name	string	yes	

`iql:id` Identifier to uniquely identify the corpus within the entire query.

`iql:name` The identifier used by the query engine's manifest registry for the corpus.

2.4 Data

Allows to embed binary data in the query and make it usable from within constraint expressions via a designated reference.

Attributes of `iql:Data`:

Attribute	Type	Required	Default
id	string	yes	
name	string	yes	
content	string	yes	
codec	string	no	hex
checksum	string	no	
checksumType	enum	no	

`iql:id` Identifier to uniquely identify the corpus within the entire query.

`iql:name` The identifier used for the expression (3.6) which can be used to reference the binary payload from within query constraints.

`iql:content` The actual content of the payload in textual form. How to properly convert the textual form to a binary stream is defined by the `iql:codec` attribute.

`iql:codec` Specifies the mechanism of converting the `iql:content` data into an actual binary stream. If left empty, defaults to `hex`.

`iql:checksum` Optional hex-string of the checksum to check the `iql:content` against.

`iql:checksumType` Defines the algorithm for computing the checksum. Currently only `MD5` is supported as legal value.

2.5 Element

Abstract base type for all logical and/or structural units that can be matched against content of a target corpus.

Attributes of `<Element>`:

Attribute	Type	Required	Default
id	string	yes	
consumed	Boolean	no	false

iql:id Identifier to uniquely identify the element within the entire query.

iql:consumed Signals that the element has already been *used up* in the context of a partial query evaluation. An element that has been consumed can safely be ignored in the further evaluation of the query. Note that this state can be propagated according to the following rules:

- A iql:Node(2.5.2) can be marked as consumed if its iql:constraint is marked as solved and its match count satisfies the iql:quantifiers requirement. Note that cross-referencing constraints can only be considered solved when all other aspects of the involved elements support the consumed state.
- A iql:TreeNode(2.5.3) can be marked as consumed if above conditions are met and all nested iql:children are marked consumed.
- An iql:Edge(2.5.4) is considered consumed when both its terminals are consumed and the same conditions regarding its iql:constraint are fulfilled as mentioned above.
- A iql:ElementDisjunction(2.5.5) is considered consumed if at least one of its iql:alternatives has been marked consumed.

2.5.1 Node Set

Wrapper around a list of iql:Element (2.5) instances to group them for either nesting or disjunction.

Extends iql:Node(2.5.2).

Attributes of iql:NodeSet:

Attribute	Type	Required	Default
nodeArrangement	enum	no	unspecified

iql:nodeArrangement Defines what kind of order or arrangement should be assumed between the elements in this set. Legal values are unspecified, ordered (matched elements must occur in exactly the order specified in this set but need not form a continuous span) or adjacent (matched elements must form a continuous span).

Nested Elements of iql:NodeSet:

Element	Type	Required
nodes	array of <Element> (2.5)	no

iql:nodes List of nested <Element> instances. Legal types depend on the context in which this set is being used.

2.5.2 Node

Logical unit for sequence or graph matching in a target corpus. May contain local constraints and can also be quantified.

Extends <Element>(2.5).

Attributes of iql:Node:

Attribute	Type	Required	Default
label	string	no	

`iql:label` Identifier to bind the node through a previously defined `iql:Binding` (2.1) declaration.

Nested Elements of `iql:Node`:

Element	Type	Required
constraint	`<Constraint>` (2.2)	no
quantifiers	array of `iql:Quantifier` (2.13)	no

`iql:constraint` Optional local constraint to be matched against the content of potential target candidates during query evaluation.

`iql:quantifiers` Optional quantifiers to define the multiplicity of matches of this node required for a positive evaluation. Multiple quantifiers behave disjunctively. Note that IQL defines some restrictions on the legal combinations of quantifiers: The all-quantifier (`*` or `all`) and not-quantifier (`!` or `not`) can only be used in isolation, all other quantifiers can be combined in disjunctive fashion.

2.5.3 Tree Node

Extension of the simple `iql:Node` type (2.5.2) to add implicit hierarchical constraints related to dominance within tree structures.

Extends `iql:Node`(2.5.2).

Nested Elements of `iql:TreeNode`:

Element	Type	Required
children	instance of `<Element>` (2.5)	no

`iql:children` Optional nested nodes or node alternatives.

2.5.4 Edge

Specialized element extension to query structural information in graphs.

Extends `<Element>`(2.5).

Attributes of `iql:Edge`:

Attribute	Type	Required	Default
label	string	no	
edgeType	enum	yes	

`iql:label` Identifier to bind the edge through a previously defined `iql:Binding` (2.1) declaration.

`iql:edgeType` The type specification for this edge, primarily a directionality information. Legal values are `simple`, `one-way` or `two-way`.

Nested Elements of `iql:Edge`:

Element	Type	Required
constraint	`<Constraint>` (2.2)	no
source	`iql:Node` (2.5.2)	yes
target	`iql:Node` (2.5.2)	yes

`iql:constraint` Optional local constraint to be matched against the content of potential target candidates during query evaluation.

`iql:source` Source node declaration.

`iql:target` Target node declaration.

For complex graph declarations multiple nodes can be defined having the same `iql:label`. The evaluation engine will treat them as being the same node. Note however, that at most **one** node per label is allowed to declare a local `iql:constraint` attribute!

2.5.5 Element Disjunction

Allows declaration of multiple alternative element definitions. When evaluating the query, each such alternative that is matched successfully will cause this element declaration to evaluate positively.

Extends `<Element>`(2.5).

Nested Elements of `iql:ElementDisjunction`:

Element	Type	Required
alternatives	array of `iql:Element` (2.5)	yes

`iql:alternatives` The alternative element declarations, each of which constitutes a legal match for this element declaration. Must not contain less than 2 elements!

2.6 Expression

Wraps the textual form of an arbitrarily complex IQL expression, which can be a formula, literal, method invocation, a combination of those or a great many other types of expressions. For more details see Section 3.6.

Attributes of `iql:Expression`:

Attribute	Type	Required	Default
content	string	yes	
resultType	string	no	

`iql:content` The textual form of the expression. Must be valid according to the specifications in Section 3.6.

`iql:resultType` An optional specification regarding the return type of the expression. Redundant when the expression is used as a constraint, as those are required to always evaluate to a Boolean result value anyway.

2.7 Group

Provides a mechanism to collect successful matches into dedicated groups, either for result visualization or use in further result processing.

Attributes of `iql:Group`:

Attribute	Type	Required	Default
id	string	yes	
label	string	yes	

`iql:id` Identifier to uniquely identify the group declaration within the entire query.

`iql:label` Label (ideally human readable) to be used for referencing this group in subsequent result processing or for generating textual result reports.

Nested Elements of `iql:Group`:

Element	Type	Required
groupBy	`iql:Expression` (2.6)	yes
filterOn	`iql:Expression` (2.6)	no
defaultValue	`iql:Expression` (2.6)	no

`iql:groupBy` The mandatory expression used to extract the value from matches based on which the actual grouping occurs.

`iql:filterOn` Optional mechanism to exclude certain matches from being used for grouping.

`iql:defaultValue` If matches cannot produce a valid value for grouping but should still be included in the process, this optional field provides the means of declaring a kind of "fall back" group. Be aware of potential overlap in groups when using default values that are not distinct from the regular grouping results.

2.8 Import

To allow for flexible integration of macro definitions or bigger language extensions, IQL provides an optional section in the query 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 providing it's unique name and telling the engine whether or not the import is to be considered optional.

Attributes of `iql:Import`:

Attribute	Type	Required	Default
id	string	yes	
name	string	yes	
optional	Boolean	no	false

`iql:id` Identifier to uniquely identify the import within the entire query.

`iql:name` The original name of the extension to be added.

`iql:optional` Defines whether or not the referenced extension is optional. Non-optional imports that cannot be resolved to an actual extension during the query evaluation phase will cause the entire process to fail.

2.9 Lane

Lanes serve as a means of splitting queries for a single corpus stream into multiple logical subqueries that target different structural and/or logical layers, e.g. multiple syntactic analyses for the same source text.

Attributes of `iql:Lane`:

Attribute	Type	Required	Default
id	string	yes	
name	string	yes	
alias	string	no	
laneType	enum	yes	

`iql:id` Identifier to uniquely identify the lane within the entire query.

`iql:name` The unique identifier of the item layer or structure layer that serves as target for this lane.

`iql:alias` If items of this lane in their entirety are meant to be used as part of query expressions inside this field holds the label used for the respective member variable. It is recommended to keep the chosen `alias` close to the original `name` to avoid confusion.

`iql:laneType` The type of structure this lane is meant to match, effectively defining the basic complexity class for evaluation. legal values are `sequence`, `tree` and `graph`. Note that the initial evaluation engine for IQL does not support the `graph` type!

Nested Elements of `iql:Lane`:

Element	Type	Required
elements	`<Element>` (2.5)	yes

`iql:elements` The structural constraints to be used for evaluation of this lane.

2.10 Layer

Every layer selector either references an entire subgraph of the corpus' member-graph directly or constructs a partial selection as part of a `iql:Scope` (2.18). When using the first approach, an item layer 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, scopes allow to create a scope that spans an exactly specified collection of layers. If multiple layer selectors are defined, up to one can be declared as "primary" to represent the granularity of returned items for the search or scope. In case no layer is explicitly marked as "primary", the one specified by the corpus or context will be used for that role by default.

Attributes of `iql:Layer`:

Attribute	Type	Required	Default
id	string	yes	
name	string	yes	
alias	string	no	
primary	Boolean	no	false
allMembers	Boolean	no	false

`iql:id` Identifier to uniquely identify the layer within the entire query.

`iql:name` Identifier used to reference the layer within its host corpus.

`iql:alias` Optional identifier to rename the layer for referencing within the query.

`iql:primary` Signals that the layer is intended to act as the primary layer in the query or scope and as such defines the level of granularity for obtaining chunks in the corpus.

`iql:allMembers` When this layer definition is used inside a `iql:Scope` (2.18), effectively adds the entire member-subgraph of this layer to the scope. This property is redundant when the layer is part of the regular `iql:layers` declaration in a `iql:Stream` (2.20), as in that case all member subgraphs for each layer are already being added to the global scope!.

2.11 Payload

Every payload encapsulates all the (processed) query constraints to be evaluated against a single stream of corpus data.

Attributes of `iql:Payload`:

Attribute	Type	Required	Default
id	string	yes	
name	string	no	
queryType	enum	yes	
queryModifier	enum	no	

`iql:id` Automatically generated identifier to uniquely identify the payload within the entire query.

`iql:name` Custom identifier to uniquely identify the payload within the entire query. This attribute is deprecated but currently being kept to shift its use case.

`iql:queryType` The overall type of query strategy to be applied for this query payload. Legal values are `all` (returns the entire corpus and disallows any kind of constraint, leaving only the `iql:Result` (2.16) declaration as option to modify the result volume), `plain` (disabling any kind of structural constraints/lanes), `singleLane` and `multiLane`. The last two values dictate the minimal/maximal number of `iql:Lane` definitions in this payload.

`iql:queryModifier` Allows to limit the number of times an individual units-of-interest (UoIs) will be returned in the result. Supported values are `first`, `last` and `any`. The specific semantics of this modifier are described in more details in Section 3.7.

Nested Elements of `iql:Payload`:

Element	Type	Required
bindings	array of `iql:Binding` (2.1)	no
lanes	array of `iql:Lane` (2.9)	no
filter	`iql:Constraint` (2.2)	no
constraint	`iql:Constraint` (2.2)	no

`iql:bindings` Optional collection of bindings used within this payload. Note that member variables inside constraints or structural query elements will not resolve unless previously bound to corpus members.

iql:lanes If `iql:queryType` is set to `singleLane` or `multiLane`, this array is expected to hold either exactly 1 or at least 2 `iql:Lane` declarations that define structural constraint for the evaluation.

iql:filter If `iql:queryType` is set to anything other than `plain`, this constraint expression allows to filter contextual UoIs prior to the actual structural matching.

iql:constraint If `iql:queryType` is set to `plain`, this is expected to contain the basic constraints for matching candidates. In any version involving `iql:Lane` declarations, global constraints can be defined here as a means of implementing complex query features that are tested once the lanes have produced preliminary result candidates.

2.12 Property

Allows customization of the evaluation process by changing parameters or switching certain features on/off.

Attributes of `iql:Property`:

Attribute	Type	Required	Default
key	string	yes	
value	string	no	

iql:key The identifier of the targeted parameter or switch. The evaluation engine might report unknown keys as errors.

iql:value The actual value to apply to the specified property in case it is not a switch.

2.12.1 Switches

For increased flexibility, IQL supports a collection of switches to turn certain optional features on or off when needed. Switches are static and cannot be changed for the active query evaluation once set. 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 those switches! Currently supported switches are shown in Table 1.

2.13 Quantifier

Specifies the multiplicity of an associated `<Element>` (2.5).

Attributes of `iql:Quantifier`:

Attribute	Type	Required	Default
quantifierType	enum	yes	
value	integer	no	
lowerBound	integer	no	
upperBound	integer	no	

iql:quantifierType Defines how to interpret the other attributes. Legal values are `all` (universal quantification), `exact`, `atMost` (0..n), `atLeast` (n+), `range` (n..m).

iql:value Target or limit value when `iql:quantifierType` is set to `exact`, `atMost` or `atLeast`.

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 3.6.12.
iql.string2bool.off	Deactivates the interpretation of strings as Boolean values as described in Section 3.7.
iql.int2bool.off	Deactivates the interpretation of integers as Boolean values as described in Section 3.7.
iql.float2bool.off	Deactivates the interpretation of floating point numbers as Boolean values as described in Section 3.7.
iql.obj2bool.off	Deactivates the interpretation of arbitrary objects as Boolean values as described in Section 3.7.
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 3.6.4 to be 0-based.
iql.warnings.off	Deactivates all warnings, potentially resulting in confusing results if there are mistakes in the query.
iql.parall.off	Forces the query evaluation engine to run single-threaded. This does however only affect the actual matcher, not additional. modules such as monitoring or item caches

Table 1: Currently supported switches in IQL and their explanations.

`iql:lowerBound` Used for `range` quantification to define the minimum multiplicity.

`iql:upperBound` Used for `range` quantification to define the maximum multiplicity.

2.14 Query

Encapsulates all the global configuration and extension of the query engine, as well as shared embedded data. Each query contains at least one `iql:Stream` declaration that in turn holds the actual query payload with constraints for the matching process.

Attributes of `iql:Query`:

Attribute	Type	Required	Default
id	string	yes	
dialect	string	no	1.0

`iql:id` Identifier for the query, chosen by the client. In more complex (asynchronous) query workflows this id is used to map answers and results to the correct query.

`iql:dialect` Specifies which basic version of IQL to use. The initial version of IQL is "1.0" and by leaving the dialect part of a query blank the engine will default to this

initial version.

Nested Elements of `iql:Query`:

Element	Type	Required
imports	array of `iql:Import` (2.8)	no
setup	array of `iql:Property` (2.12)	no
embeddeData	array of `iql:Data` (2.4)	no
streams	array of `iql:Stream` (2.20)	yes

`iql:imports` Defines extensions to be applied to the evaluation engine prior to actual query evaluation.

`iql:setup` Allows to configure the core evaluation engine or already defined extensions in a simple manner.

`iql:embeddeData` Binary data to be used in the evaluation process, such as audio or video fragments.

`iql:streams` Corpus data streams to be queried. In the initial version, the engine only supports single-stream querying!

2.15 Reference

Models references usable from within query expressions for accessing corpus members or variables.

Attributes of `iql:Reference`:

Attribute	Type	Required	Default
id	string	yes	
name	string	yes	
referenceType	enum	yes	

`iql:id` Identifier to uniquely identify the reference within the entire query.

`iql:name` The local identifier to be used for addressing this reference. Note that this is the bare name without any type-specific prefixes (such as '`$`' for members, cf. Section 3.5).

`iql:referenceType` Specifies the nature of this reference. Legal values are `reference`, `member` or `variable`.

2.16 Result

Encapsulates all the information on result processing and preparation.

Attributes of `iql:Result`:

Attribute	Type	Required	Default
limit	integer	no	
percent	Boolean	no	false

`iql:limit` Optional limitation on the total size of the result to be returned. If the `iql:percent` flag is not set to `true`, this number is in reference to the units provided by the query's primary item layer.

`iql:percent` If set to `true` the value defined in `iql:limit` is treated as a integer percentage value in the interval 1 to 99, with boundaries included.

Nested Elements of `iql:Result`:

Element	Type	Required
resultTypes	array of enum	yes
resultInstructions	array of `iql:ResultInstruction` (2.17)	no
sortings	array of `iql:Sorting` (2.19)	no

`iql:resultTypes` Defines the result format or type the engine should return data in. At least one result type must be declared and the engine can also be instructed to return the results in multiple formats simultaneously. In the first iteration only `kwic` (keyword-in-context) and `custom` (as a placeholder for the raw corpus members) are supported.

`iql:resultInstructions` Optional collection of additional processing instructions to generate (textual) result reports.

`iql:sortings` Allows to sort matches before generating result reports.

2.17 Result Instruction

Currently unused dummy for declaring post-processing instructions on the query result to perform conversions and/or tabular calculations.

2.18 Scope

Very detailed vertical filtering of the layers available in a query.

Attributes of `iql:Scope`:

Attribute	Type	Required	Default
id	string	yes	

`iql:id` Identifier to uniquely identify the scope within the entire query.

Nested Elements of `iql:Scope`:

Element	Type	Required
layers	array of `iql:Layer` (2.10)	yes

`iql:layers` The layer members of this scope.

2.19 Sorting

Defines a single rule for sorting query results based on an arbitrarily complex expression.

Attributes of `iql:Sorting`:

Attribute	Type	Required	Default
order	enum	yes	

`iql:order` Hint on sorting direction, legal values are `asc` or `desc` for ascending or descending order, respectively.

Nested Elements of `iql:Sorting`:

Element	Type	Required
expression	`iql:Expression` (2.6)	yes

`iql:expression` The actual sorting expression. It can use any (member) reference or variable available in the query to compute its result and must return a type that is comparable to allow stable sorting. Per default any of the primitive numerical types (`int` or `float`), the text type `string` and any member of the ICARUS2 framework implementing the `java.lang.Comparable` interface can be used as return type.

2.20 Stream

A stream encapsulates all the information and query constraints to extract, evaluate and prepare data from a single corpus. Note that many of the attributes and/or elements below are marked as optional, but the following restrictions are in effect:

- Either `iql:rawPayload` or `iql:payload` must be provided by the client.
- Either `iql:layers` or `iql:scope` must be provided to define the granularity of data being loaded for evaluation.

Attributes of `iql:Stream`:

Attribute	Type	Required	Default
id	string	yes	
primary	Boolean	no	false
rawPayload	string	no	
rawGrouping	string	no	
rawResult	string	no	

`iql:id` Identifier to uniquely identify the stream within the entire query.

`iql:primary` Flag to indicate that the primary layer of this stream is meant to be used as primary layer of the entire search result. Only one stream can declare this property and it primarily dictates the order of result elements in a multi-stream query or which stream is allowed to dictate sorting.

`iql:rawPayload` The textual (raw) form of the payload for this stream, i.e. all the constraints and structural query content.

`iql:rawGrouping` The textual (raw) grouping definitions to be applied for results of this stream.

`iql:rawResult` The textual (raw) result configuration and post-processing instructions for this stream.

Nested Elements of `iql:Stream`:

Element	Type	Required
corpus	`iql:Corpus` (2.3)	yes
layers	array of `iql:Layer` (2.10)	no
scope	`iql:Scope` (2.18)	no
payload	`iql:Payload` (2.11)	no
grouping	array of `iql:Grouping` (2.7)	no
result	`iql:Result` (2.16)	yes

`iql:corpus` The corpus to extract data from.

`iql:layers` Vertical filtering to be applied to the corpus prior to actual query evaluation.

`iql:scope` Another and more fine-grained form of vertical filtering that allows for more precise selection of layers to be part of this stream's data.

`iql:payload` The processed form of `iql:rawPayload`.

`iql:grouping` The processed form of `iql:rawGrouping`.

`iql:result` The processed form of `iql:rawResult`.

3 Inner IQL ELements

Certain parts of an IQL query can be defined in *raw* form, that is, in a keyword-driven formal language. During the first phase of query evaluation they get (partly) translated into their respective JSON-LD counterparts described in Section 2 (unless of course the query or query fragments are provided fully processed). This section defines the syntax and additional rules for those raw statements. Note that the textual form of all following IQL elements is expected to be encoded in UTF-8, so no special escape mechanisms are needed for unicode content.

3.1 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, and while camel-cased versions are technically permitted, it is highly discouraged to use them:

ADJACENT	DISTINCT	FROM	ODD
ALL	DO	GROUP	OMIT
AND	EDGES	HAVING	ON
ANY	END	IN	OR
AS	EVEN	LABEL	ORDER
ASC	FALSE	LANE	ORDERED
BY	FILTER	LAST	RANGE
COUNT	FIND	LIMIT	STEP
DEFAULT	FIRST	NOT	TRUE
DESC	FOREACH	NULL	WITH

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

boolean	int	float	string

3.2 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.

3.3 Literals

Literals are statically-typed fixed-value expressions in IQL. They are parsed only once during the initial processing part of a query.

3.3.1 String Literals

IQL uses simple double quotes ('"' or U+0022) to define string literals. String literals may not contain any of the following symbols directly:

```
\n line break
\r carriage return
\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. At the current time there is no planned mechanism to provide additional escape support for unicode symbols, since the default encoding scheme for IQL is UTF-8.

Examples for valid string literals:

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

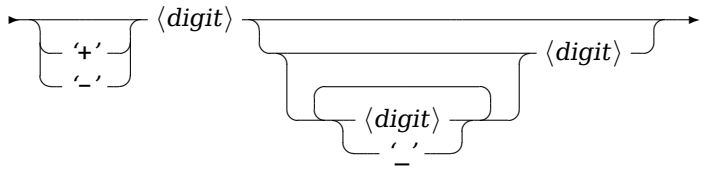
3.3.2 Boolean Literals

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

3.3.3 Integer Literals

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 (‘`0`’ to ‘`9`’) or underscore (`_`) characters. Underscore characters may only appear inside the integer literal, never at the beginning or end (not counting the sign symbol).

Grammar Snippet 1
⟨*integerLiteral*⟩:



Examples for valid (signed) integer literals:

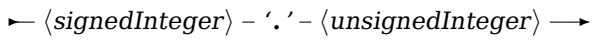
```
1
+123
-123
1_000_000
-99_000000_0
```

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.

3.3.4 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.

Grammar Snippet 2
⟨*floatingPointLiteral*⟩:



Examples for valid (signed) floating point literals:

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

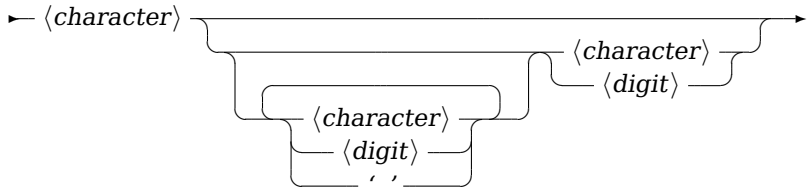
While many languages offer to express floating point literals in the scientific notation with explicit exponent declaration, we do not include this in the initial draft of IQL.

3.4 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.

Grammar Snippet 3

⟨*identifier*⟩*:*



Examples for valid identifiers:

```
x
myIdentifier
x1
x_1
x__1
x321
some_random_id
someRandomId002
random_2_4
notTheBest_____________example
```

Identifiers are limited in length by the engine to a total of 255 characters. This is a purely arbitrary choice to keep queries readable and not subject to any technical limitations.

3.5 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 namespace 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 additionally 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`. Table 2 provides a compact overview of the available identifiers and their capabilities/features.

Type	Prefix	Example	Scope	Fixed[1]	Final	Re-Assign
Reference	none	`max()`	global	X	X	
Variable	`@`	`@myVar`	limited	(X)		X
Member	`$`	`$token`	limited	X	(X)	

Table 2: Identifier types available in IQL and their properties.

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.

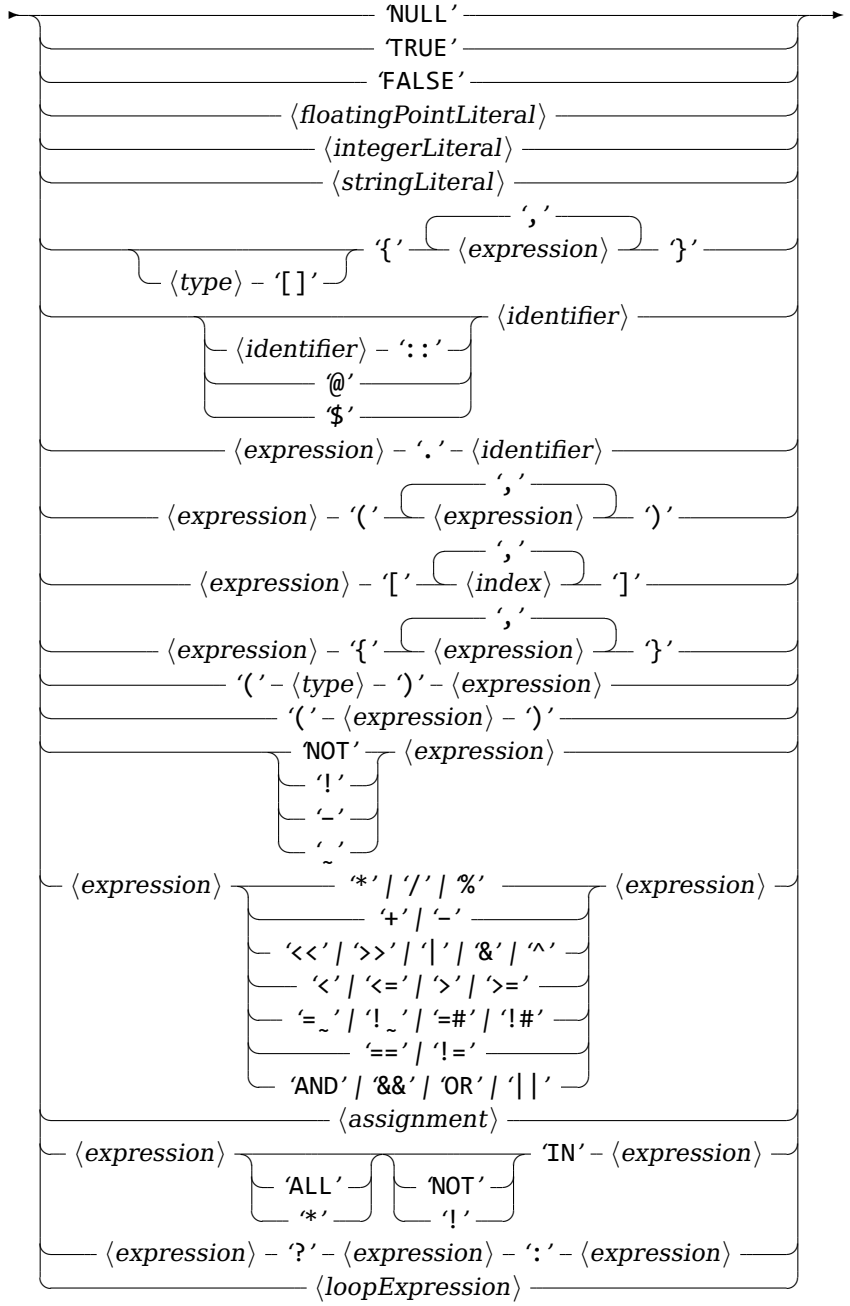
Member identifiers are final in the sense that they cannot be re-assigned explicitly but will be implicitly for every iteration of the query on a new part of the corpus. For example, above `$token` member will point to a new token object every time the inner constraint parts of the query are evaluated.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

3.6 Expressions

Expressions are the foundation of every query. Each expression has a (usually fixed) result type and evaluates to a value of that type. They can take any of the following forms:

Grammar Snippet 4

⟨*expression*⟩*:*



3.6.1 Primary Expressions

Any literal of types `boolean`, `string`, `int` or `float` can serve as a primary expression of that type. See Section 3.3 for examples and a more detailed specification of the various types of literals in IQL.

3.6.2 Path Expressions

For navigating hierarchically structured object graphs or namespaces, expressions can take the form of paths, consisting of a original expression, a dot as separator and finally

an identifier that denotes the path element or “field” within the context of whatever the original expression returned.

Grammar Snippet 5 (⟨*path*⟩)
⟨*path*⟩ *::=* ⟨*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.

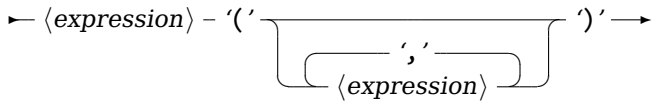
3.6.3 Method Invocation

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

Grammar Snippet 6 (⟨*method*⟩)
⟨*method*⟩ *::=* ⟨*expression*⟩ ‘**(**’ ⟨*arguments*⟩*?* ‘**)**’

⟨*arguments*⟩ *::=* ⟨*expression*⟩ *(‘**,**’* ⟨*expression*⟩*)**

⟨*method*⟩*:*



Examples:

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

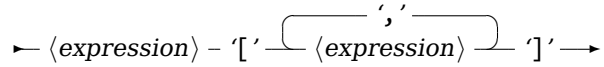
3.6.4 List Access

Lists or arrays are accessed by an expression pointing to the list or array object itself and an index expression in square brackets indicating the position(s) of the desired element(s) within the array. Note that the index or indices expression must evaluate to integer values within `int` space. Positive values indicate the position beginning from the start of the 0-based array, 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.

Grammar Snippet 7 (⟨*array*⟩)
⟨*array*⟩ *::=* ⟨*expression*⟩ '[' ⟨*indices*⟩ ']'

⟨*indices*⟩ *::=* ⟨*expression*⟩ *(',' ⟨expression⟩)**

⟨*method*⟩*:*



Examples:

```
myArray[1]
myArray[-1]
myArray[-myArray.length] // same as myArray[0]
complexArray[1][2][3]
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 "myContainer.getItemAt(someIndex)" can be used the same as any regular array with the expression "myContainer[someIndex]".

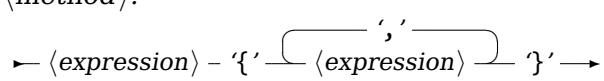
3.6.5 Annotation Access

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:

Grammar Snippet 8 (⟨*annotation*⟩)
⟨*annotation*⟩ *::=* ⟨*expression*⟩ '{' ⟨*keys*⟩ '}'

⟨*keys*⟩ *::=* ⟨*expression*⟩ *(',' ⟨expression⟩)**

⟨*method*⟩*:*



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 3.6.12) 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"}

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

3.6.6 Type Cast

Expressions in IQL are automatically cast to matching types according to the actual consumer's needs (unless this feature gets deactivated via the corresponding switch, cf. Section 2.12.1). Explicit casts can be performed by preceding an expression with one of the type keywords listed above (3.1) in round brackets.

Examples:

```
(int) myValue
(int) 12345.678
(float) average(myVector)
(string) 123.456
```

3.6.7 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 the outside context.

Examples:

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

3.6.8 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') as shown in snippet 9. 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 about equality operators in Section 3.6.10. Note that methods or collections used inside the set definition are subject to the expansion rules described in Section 3.6.12. 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 wrapping 3.6.7 them and negating 3.6.9 the entire expression) 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

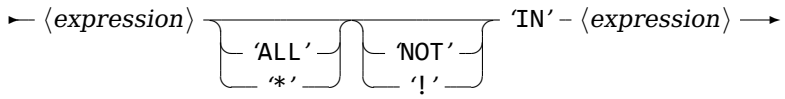
be contained in the set. The set predicate can be universally quantified with a star '*' or the `ALL` keyword in front to change the overall behavior such that the result will be `true` iff *all* of the elements are contained in the set (or none of them are, if the set predicate is directly negated).

Grammar Snippet 9 (⟨*set_predicate*⟩)
⟨*set_predicate*⟩ ::= ⟨*expression*⟩ ⟨*all*⟩? ⟨*not*⟩? 'IN' ⟨*expression*⟩

⟨*all*⟩ ::= 'ALL' | '*'

⟨*not*⟩ ::= 'NOT' | '!'

⟨*set_predicate*⟩:



Examples:

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

3.6.9 Unary Operation

IQL only allows four unary operators to be used directly in front of an expression, the exclamation mark '!' and the `NOT` keyword for Boolean negation, the minus sign '-' for negating numerical expressions and the '~' symbol of bitwise negation of integer numbers.

Examples:

```
!someBooleanFunction()
NOT someBooleanValue
-123
-myNumericalFunction()
~123
~myIntegerFunction()
```

3.6.10 Binary Operation

Binary operations between two expressions take the following simple form:

Grammar Snippet 10 (⟨*binary_op*⟩)
⟨*binary_op*⟩ ::= ⟨*expression*⟩ ⟨*operator*⟩ ⟨*expression*⟩

Binary operators follow an explicit hierarchy, listed in Table 3 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

Table 3: Binary operators available in IQL and their hierarchical order.

Basic Numerical Operations Basic numerical operations follow the standard mathematical rules for priorities. While the basic numerical types (`int`, `float`) 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.

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 3.6.6 to match the other. If value expansion 3.6.12 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.

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 and may be subject to case conversions (2.12.1).

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 (2.12.1) are planned to allow finer control over regex details.

Examples:

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

Equality Equality checks follow the same basic conditions as ordered comparisons (3.6.10), 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.

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
```

3.6.11 Ternary Operation

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

Grammar Snippet 11 (⟨*ternary*⟩)
⟨*ternary*⟩ *::=* ⟨*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 final 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"
```

3.6.12 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 3D 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()]".

Type	Condition	Value
`string`	empty or null	`false`
`int`	`0`	`false`
`float`	`0.0`	`false`
any object	`null`	`false`

Table 4: Rules for converting arbitrary values or objects in a query to Boolean values.

3.7 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 certain evaluations as syntactic sugar, listed in Table 4. Note that those conversions are only active if the respective switches to disable them (2.12.1) have not been set.

3.8 Payload Structure

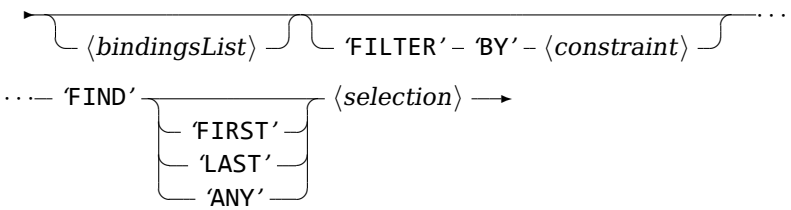
The Payload section in IQL consists of either the sole `ALL` keyword or a selection statement (3.8.4) with optional binding (3.8.3) definition and filter constraints (3.8.1) preceding it. If the `ALL` keyword is used, no constraints whatsoever can be defined and the engine is instructed to return the entire target corpus. In this case the only way of restricting results is by using the `iql:Result` section (2.16) of a query.

Grammar Snippet 12 (⟨*payload*⟩)

⟨*payload*⟩ ::= 'ALL'
| ⟨*bindingsList*⟩? ('FILTER' 'BY' ⟨*constraint*⟩)? 'FIND' ⟨*modifier*⟩? ⟨*selection*⟩

⟨*modifier*⟩ ::= 'FIRST' | 'LAST' | 'ANY'

⟨*payload with content*⟩:



3.8.1 Filter Constraints

For complex (i.e. structural) queries, IQL offers a way of filtering the UoIs before they are processed by the matchers for sequence, tree or graph structures (cf. Sections 3.8.8 to 3.8.10). A dedicated `FILTER BY` section in the query payload preceding the actual structural constraints is available to define filtering rules that have to evaluate to `true` for a UoI to be considered for actual matching. Constraints within a filtering rules have only access to general properties of the UoIs, such as sentence length, tree height or similar information. The do **not** have access to bound member variables, apart from those defined for the top-level members of lanes (3.8.5)! Note that filter constraints are **not** compatible with flat constraints (3.8.6) as they both essentially fill the same function and flat constraints take precedence.

3.8.2 Match Modifiers

Per default, the search in IQL is expected to be exhaustive, i.e. the evaluation engine will attempt to find all of the instances in a target corpus that match the query constraints, potentially reporting individual UoIs (such as sentences) multiple times if they contain several occurrences. For instance, the sentence "The dog chased the rabbit down the hill." will be treated as tree entries in the result if the query was only meant to find instances of the lemma "the". This default behavior can be adjusted to only return each UoI no more than once by using one of the modifiers (`FIRST`, `LAST`, `ANY`) listed in snippet 12. The semantics of the `ANY` modifier are such that the engine may freely pick any one match within a UoI. Note however, that to support reproducible search results, repeated evaluations of the same query on a corpus are still required to yield the same instances here. The exact semantics of `FIRST` and `LAST` are depending on the type of structural constraints used in the payload, but generally are based on the natural order of items within the corpus (typically this is the flow of words in a text). The evaluation behavior for them is subsequently covered in Sections 3.8.8 to 3.8.10.

3.8.3 Bindings

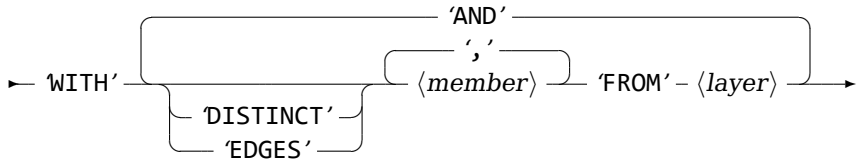
A binding is a collection of member references (3.5) that get declared to belong to a certain member type or part of the corpus. The `DISTINCT` keyword enforces that multiple bound member references in this binding do **not** match the same target. Depending on the local constraints used in the query, 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. Additionally the `EDGES` keyword signals that the bound members of a structure are actually edges. In this case using `DISTINCT` is redundant, as bound edges are implicitly assumed to be distinct when matching.

Grammar Snippet 13 (Bindings)
⟨*bindingsList*⟩ ::= 'WITH' ⟨*binding*⟩ *('*AND*'* ⟨*binding*⟩*)**

⟨*binding*⟩ := ⟨*option*⟩*?* ⟨*member*⟩ *('*,*'* ⟨*member*⟩*)** 'FROM' ⟨*layer*⟩

⟨*option*⟩ := 'DISTINCT' | 'EDGES'

⟨*bindingsList*⟩*:*



Raw binding definitions in the payload are parsed and stored in their JSON counterpart (`iql:Binding`, 2.1) during query processing.

3.8.4 Selection Statement

Constraints are further divided into local constraints as part of node or edge definitions and global ones (with the `HAVING` keyword). Local constraints are obligatory and define

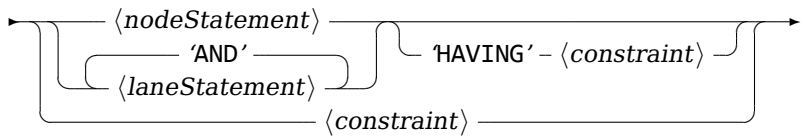
the basic complexity of the query (flat, tree or graph). They also introduce certain limitations on what can be expressed or searched (e.g. a "flat" local constraints declaration will not provide implicit access to tree information). However, global constraints can introduce arbitrary constraints or relations 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.

Grammar Snippet 14 (⟨*selectionStatement*⟩)
⟨*selectionStatement*⟩ *::=* ⟨*constraint*⟩
| *(*⟨*nodeStatement*⟩ *|* ⟨*laneStatementsList*⟩*)* *(*'HAVING' ⟨*constraint*⟩*)?*

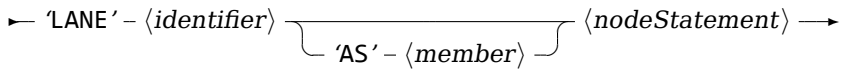
⟨*laneStatementsList*⟩ *::=* ⟨*laneStatement*⟩ *(*'AND' ⟨*laneStatement*⟩*)**

⟨*laneStatement*⟩ *::=* 'LANE' ⟨*identifier*⟩ *(*'AS' ⟨*member*⟩*)?* ⟨*nodeStatement*⟩

⟨*selectionStatement*⟩*:*



⟨*laneStatement*⟩*:*



3.8.5 Lanes

Lane statements can be used to extract information from concurrent structures that exist for the UoI of the payload. Each lane statement is introduced by the LANE keyword and an identifier that matches the name or alias of a layer in the outer query definition (cf. snippet 14). Optionally the source layer of a lane can also be assigned a member variable (3.5) so that it can be explicitly referenced in the payload.[2] During query processing raw lane statements will be parsed into `iql:Lane` objects (2.9).

Constraints are further divided into local constraints as part of node or edge definitions 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 or relations 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.

[2]This is particularly useful when using the global constraints to compare content of different lanes. Imagine for instance a query that searches for a certain syntactic construct `C` to be present in two concurrent parse trees `A` and `B`, but will only consider sentences where `C` is embedded deeper inside `A` compared to its embedding depth in `B`.

3.8.6 Flat Constraints

Flat constraints provide no extra helpers to declare structural properties of the query. They consist of arbitrary basic constraints Section 3.7 and disallow both global constraints (3.8.11) and filter constraints (3.8.1).

a word on the differences between flat and global constraints

3.8.7 Structural Constraints

IQL provides several classes of structural constraints that each feature distinctive syntax features to express structures of increasing complexity. Those structures are sequences (3.8.8), trees (3.8.9) and graphs (3.8.10). They all get explained in more detail in their respective sections, but the syntactic basics for all of them will be defined here. To simplify the overall IQL grammar, a general syntax exists for the declaration of nodes (and edges). This general form honors the aspects specific to each of those structure types, but generally over-generates and only some of its features are actually applicable in concrete use cases. Snippet 15 shows the full syntax for defining structural constraints in IQL.

Grammar Snippet 15 (⟨*nodeStatement*⟩)

⟨*nodeStatement*⟩ ::= '{' ⟨*nodeStatement*⟩ '}'
| ⟨*nodeArrangement*⟩? ⟨*node*⟩+
| ⟨*element*⟩ (',' ⟨*element*⟩)*
| ⟨*nodeStatement*⟩ 'or' ⟨*nodeStatement*⟩

⟨*nodeArrangement*⟩ ::= 'ORDERED' | 'ADJACENT'

⟨*node*⟩ ::= ⟨*quantifier*⟩? '[' ⟨*memberLabel*⟩? ⟨*constraint*⟩? ⟨*nodeStatement*⟩? ']'

⟨*quantifier*⟩ ::= ⟨*simpleQuantifier*⟩ ('|' ⟨*simpleQuantifier*⟩)*
| '<' ⟨*simpleQuantifier*⟩ ('|' ⟨*simpleQuantifier*⟩)* '>'

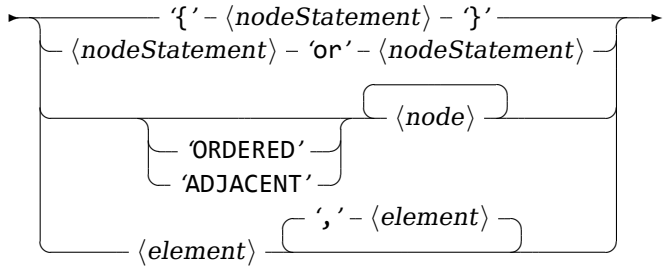
⟨*simpleQuantifier*⟩ ::= ('ALL' | '*')
| ('NOT' | '!')
| ⟨*digits*⟩ ('+' | '-')?
| ⟨*digits*⟩ '..' ⟨*digits*⟩

⟨*memberLabel*⟩ ::= ⟨*member*⟩ ':'

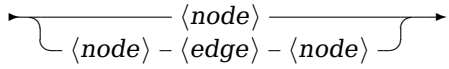
⟨*element*⟩ ::= ⟨*node*⟩ | ⟨*node*⟩ ⟨*edge*⟩ ⟨*node*⟩

⟨*edge*⟩ ::= '<--' | '-->' | '<->' | '---'
| ('<-' | '--') '[' ⟨*memberLabel*⟩? ⟨*constraint*⟩? ']' ('--' | '->')

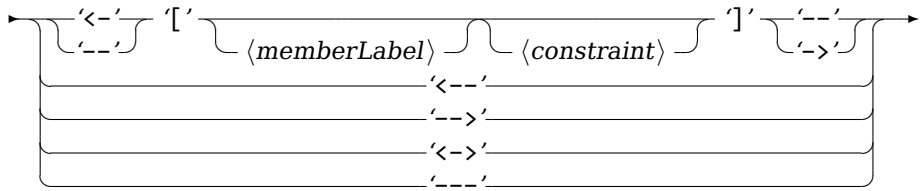
⟨*nodeStatement*⟩:



⟨*element*⟩:

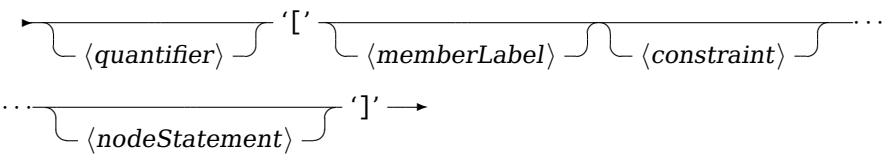


⟨*edge*⟩:



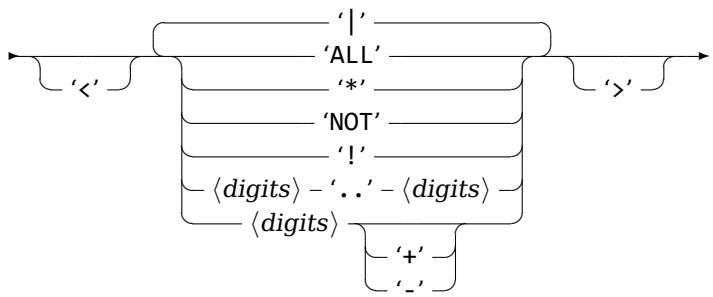
There are four (partly recursive) approaches to express node statements, i.e. grouping, node sequence, element sequence and disjunction. The distinction between node and element sequences exists to easily distinguish sequence or tree queries from graph definitions. Sequence queries do not include hierarchical structural information and as such have no use for edges. In the syntax used for tree nodes in IQL information about the incoming edge is implicitly available from every nested node and constraints related to outgoing edges are to be attached to the respective child terminals of those edges. For graphs where no simple association between nodes and edges exists, there is a necessity to have explicit edge declarations available for querying. As such the ⟨*element*⟩ rule in snippet 15 is a placeholder that can be filled with either node or edge declarations.

Nodes IQL uses square brackets ([and]) to mark individual nodes.
⟨*node*⟩:



Declaring a node in a structural constraint implicitly marks it as existentially quantified.

Additionally, nodes can be **explicitly quantified** with an arbitrary combination of universal quantification, negation, explicit quantification, *at-most*, *at-least* or bounded range quantification. The following diagram simplifies the overall rules for ⟨*quantifier*⟩ to keep it compact. Albeit being shown here as unrelated to each other, the appearance of the angle brackets (< and >) before or after the actual quantifier content is restricted to either *both* of them being used (for a proper wrapping, such `<3..10 >`) or *none* of them (for plain quantifiers, such as `1|4|ALL`).
⟨*quantifier*⟩:



IQL allows multiple quantifiers to be separated by the pipe symbol '|' to express disjunc-

tion between quantifiers. This way complex constraints can be defined very neatly, such as the "all or nothing" quantification `all|not`. This quantifier combination ensures that either all targets in a certain context match the node in question, or none does.[3]

A node's inner content can optionally have an initial **member label** (identifier with a colon ':' afterwards) to link this node to a previously defined member binding (3.8.3). Such as binding restricts the type of corpus member that the node can be matched against. It also provides a point of reference that subsequent constraints (e.g. in the global constraints section, cf. Section 3.8.11) can use to access information of the target the node has been matched against. Note that cross-referencing between nodes from within local constraints (such as in `[$x:][$y: pos!=$x.pos]`) is discouraged[4] and global constraints should be used for this. This approach guarantees that by the time such cross-reference constraints (or "joins", to use database terminology) are evaluated all involved member variables will be assigned preliminary candidates.

Nodes can also optionally define **local constraints** that must evaluate to `true` for target item to be considered as result candidate. Local constraints have full access to properties and annotations of the target item currently being inspected and can take any form described in Section 3.7. Note that it is up to the evaluation engine how to optimize and potentially prune the evaluation process of constraint expressions. Correct evaluation of a conjunctive local constraints with external function calls `[$x: func1($x) && func2($x) && func3($x)]` must not rely on the premise of any particular function (`func1`, `func2` or `func3`) actually being called at all. The evaluation semantics of conjunctive Boolean concatenation allow an early determination of the final result as soon as the stable predicate of one of the inner terms evaluating to `false` is met. Therefore it is perfectly legal (and in parts expected from an efficient evaluation engine) to not evaluate the calls to `func2` and `func3` after `func1` has already caused the result to remain `false`.

If a ⟨*node*⟩ is used within a tree environment, it can also contain a **nested** ⟨*nodeStatement*⟩ declaration to define structural constraints on child nodes. The target item a node is matched against during query evaluation defines the structural context that is then in turn being used for matching the nested ⟨*node*⟩ instances.

Edges

Node Grouping Nodes (or elements) can be grouped together within curly brackets (`{` and `}`) as defined by the first ⟨*nodeStatement*⟩ rule. This is useful for either restricting the scope of modifiers or directives such as the `ADJACENT` keyword to only a selected few nodes or when expressing a disjunction. Note that a group counts as an individual node statement inside the outer scope and as such is subject to order directives defined there. However, those directives are **not** automatically inherited to the inner collection of nodes in the group, allowing for expressions such as the following node sequence:

```
ADJACENT [$x:] {[$a:][$b:]} {[$c:][$d:]} [$y:]
```

[3] Example: find all sentences that either have no word with more than five characters or all of their words have five or more characters.

[4] The reason behind this is that the ICARUS2 Query Processor (IQP) per default is not required to honor the order of nodes defined in a query or the linking relations between them when planning the automaton for evaluation. As such there is no guarantee that node `x` will have already been matched against a valid target when the cross-reference constraint inside `y` is evaluated. This would cause an error during evaluation time to occur, which in turn will abort the entire search.

This would read as “Find x immediately followed by a, later followed by b+c, later followed by d+y”. Note that the adjacency modifier does not apply to the inner sequences a+b and c+d, which are only subject to the implicit order of the sequence declaration.[5] The concept of node grouping is especially important for the tree (3.8.9) and graph (3.8.10) constraints introduced below, as by default those do not impose an a priori order of nodes.

Node Sequence Nodes usable for sequences (3.8.8) and trees (3.8.9) are defined in a simple sequence style (second ⟨*nodeStatement*⟩ rule). Instances of ⟨*node*⟩ in a sequence are defined one after another without special separator symbols. They may optionally be preceded by a ⟨*nodeArrangement*⟩ directive to guide the matching progress. Currently there are only two directives available to specify the node arrangement (ORDERED and ADJACENT), but this might increase in the future, making node grouping a very important tool for defining complex structural compositions.

Element Sequence Similar to node sequences, ⟨*element*⟩ instances can also be used in a list-style collection (third rule of ⟨*nodeStatement*⟩), but with noticeable differences: Element sequences do use a separator symbol (a simple comma ‘,’) between ⟨*element*⟩ definitions. Since IQL does not use keywords to signal the structural type to be expected in a query payload[6] this approach was necessary to easily detect the type of structure. It also hints at the second difference, that is, element sequences do not support arrangement modifiers (as ⟨*element*⟩ instances can be either nodes or edges, with the latter not being suitable for this kind of ordering) and as such can be more intuitively be understood as sets of ⟨*element*⟩ instances.

Structural Disjunction As the forth option of ⟨*nodeStatement*⟩, the disjunction of entire node statements provides a very powerful tool to express complex queries.

syntax and examples?

3.8.8 Sequence Constraints

As the most basic form of structural constraints this type is used to match sequences of items or nodes in the target corpus. Multiple nodes in a sequence declaration are required to match to items in exactly the order they are defined in (but not necessarily adjacent to each other, use the ADJACENT directive in front of a node sequence for that).

Examples:

```
[]                  // empty node
[pos=="NN"]         // node with local constraint
<2>[$x:]            // node x exactly 2 times
[$x:]<2-5>[][$y:]   // nodes x & y with 2-5 nodes in between
[$x:] ![$y:]        // node x without any node y following
ADJACENT [$x:][$y:] // node y directly following node x
[$x:] or [$y:]      // disjunction: either x OR y
{[$x:][$y:]} or [$z:] // disjunction: either group x+y OR z
```

[5] An additional ORDERED in each of the groups would make that explicit, but is redundant.

[6] An earlier draft made use of TREE and GRAPH keywords to distinguish those types from the basic node sequence, but in an effort to reduce the overall number of keywords (that users had to learn) this approach was dropped.

Sequence Matching Sequence constraints only provide a single dimension for *moving* the query sequence through the search space of the target corpus. Matching is performed greedily in order of node appearance in the query, following the direction specified by the corpus itself. If the switch to reverse direction (cf. 2.12.1) is set, then matching attempts will start from the very end of each matching context (such as a container or structure), but the order of nodes to match will remain the same. That is, in a node sequence `[$x:][$y:]` the node matched by `x` will **always** be before the node matched by `y` according to the original direction of items in the corpus. Only the direction from which the overall matching attempt will start changes with the respective switch. This also leads to a very simple and intuitive semantic for the `FIRST` and `LAST` modifier keywords: They stay true to their names and limit the returned match to either the first or last, with respect to the current direction.

Special Note: Evaluation of quantified nodes in IQL is done greedily, i.e. the engine will try to match as many instances as possible. However, since an empty node does only provide the constraint of existential quantification, the above example will find the first occurrence of pair c+d after the initial hit on pair a-b. An extension of the quantifier syntax or a change in the "at most" semantics is on the drawing board, but currently not considered to be implemented (a new keyword `inf` could potentially be used with the "at most" quantifier to create a hint to greedily expand empty nodes to the total limit instead of a "first hit" eager strategy, such as `<inf->`).

Empty nodes with quantifiers can be used as proxies to model distance constraints, as seen in above examples. Since the `ADJACENT` directive changes the behavior of an entire node sequence, some creativity can be necessary to achieve mixed cases, such as *"find an adjacent pair a+b that is later followed by another adjacent pair c+d"*. A possible (and simple) solution for this query would be the following:

```
ADJACENT [$a:][$b:] <0+>[] [$c:][$d:]
```

3.8.9 Tree Constraints

Located between mere sequences (3.8.8) and graphs (3.8.10) this type of structural constraints is meant to target tree structures, such as (but not limited to) syntax trees, coreference structures, discourse, etc. To simplify query syntax, IQL uses a similar approach as the original ICARUS project, which in turn took inspiration from PML-TQ: To signal parent-child relations, child nodes are nested within their respective parent, effectively making each node yet another scope for a sequence of child nodes. Contrary to bare sequence constraints (3.8.8) the order of (child) nodes to be matched in the corpus is **not** implicitly defined by the order of constraint nodes! Instead, the `ORDERED` or `ADJACENT` keywords need to be used explicitly to signal that a specific kind of order should be honored. Apart from this little addendum tree constraints behave basically the same as nested sequence constraints: They can be individually quantified or existentially negated, as well as grouped and linked via the `OR` keyword to expression disjunctions.

Examples:

```
[[]]              // anonymous nesting of nodes
[$x: [$y:]]       // nesting of node y inside x
[[$x:] [$y:]]     // nesting of siblings x and y
[$x: [$y: [$z:]]] // deep nesting chain
[$x: <2->[$y:]]   // at most 2 y nested inside x
[$x: ![$y:]]      // node x without any child matching y

// internal disjunction
[[$x:] or [$y:] or {[$z: <4+>[]]}]
```

Tree Matching As opposed to sequences (3.8.8), trees (and subsequently also graphs, cf. 3.8.10) offer an additional dimension of matching freedom over the bare iteration of items in a container or structure to be matched. This requires further specification of the matching process to guarantee consistent results and define basic expectations. Below explanations are primarily intended to specify the behavior in the presence of limiting modifiers (`FIRST`, `LAST`, `ANY`) in the payload, but are also of interest for the expected order of returned matches if no limiting modifiers are defined.

Consider a simple tree query `[$x: [$y:]]` where x and y are bound nodes with individual constraints. The nodes X_n or Y_n in the example tree (Fig. 1) then denote the n-th node that match the constraints of query nodes x and y, respectively.

The tree contains 3 possible matches for the query, specifically the pairs $\{X_1,Y_1\}$, $\{X_1,Y_2\}$ and $\{X_2,Y_2\}$. `[Root [A1] [A2 [X1 [A3] [Y1 [A4] [A5]] [Y2]]] [X2 [A6] [Y3]]]`

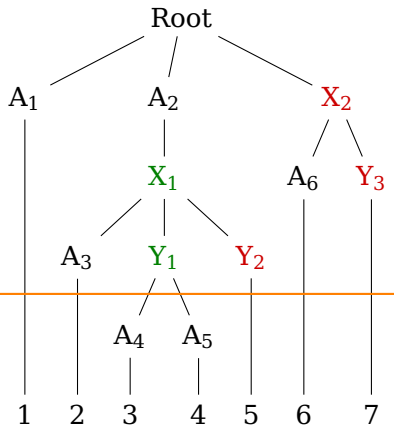


Figure 1: Example tree with highlighted hits for the query `[$x: [$y:]]`.

expand on the order of matches

3.8.10 Graph Constraints

content

3.8.11 Global Constraints

Global constraints con be any basic constraint Section 3.7 and follow after the main section of structural constraints, indicated by the `HAVING` keyword.

examples and more explanation of what is available (members, etc) + complexity!!!

Evaluation Priorities If global constraints are present, the evaluation process changes to a two-stage strategy: Matchers for the associated structural constraints produce preliminary result candidates and the global constraints are then evaluated for each such candidate. This makes global constraints both very powerful as they have access to more information compared to regular (internal or local) constraints (e.g. they already *know* that all local constraints evaluated to `true` and the exact candidates produced for structural constraints) and also very critical when it comes to performance. It can be very

tempting to construct queries such as the following one (bindings section omitted) that only matches when y is the last child of x:

```
FIND [$x: [$y:]] HAVING $x.indexOf($y) == $x.size-1
```

This will cause the structural matcher to potentially propose **all** children of x as candidates to be processed by the global constraints section. Subsequently, for a node of size N this will produce $N - 1$ candidates that are bound to fail the global constraint check. Section 4 lists several families of utility markers and functions that can be used to signal the evaluation engine that certain local constraints are to be treated as special filters. With the use of those utility markers, above query looks like the following and will be vastly more efficient to evaluate:

```
FIND [$x: [$y: isLast]]
```

Similarly global constraints are not the place to perform filtering on general properties of the current UoI, such as sentence length (use the `FILTER BY` expression for that, cf. Section 3.8.1).

3.9 Result Processing

There be dragons...

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

4 Utility Markers & Functions

The following utility features are provided by the IQP but are **not** part of the core specification. As such it is possible for engine extensions to override them, change their behavior or completely remove them if desired. They are listed here as per default they all are available and provide valuable improvements for performance and usability.

4.1 Position Markers

Every node (3.8.8) in a query has an implicit Container or Structure context that it is hosted or contained in.[7]

describe the situations possible for nodes (root, top-level, nested) with implicit/explicit parent, and then list the isFirst, isLast, etc mark-

4.2 Tree Functions

4.3 Graph Functions

[7] The ICARUS2 Corpus Modeling Framework (ICMF) specifies that each item can only be hosted (or *owned*) by a single container or derived object, but be contained within an arbitrary number of additional containers or structures.