Users' Manual i-views 6.1

Table of contents

1. Knowledge-Builder . . . 1
    1.1. Basics . . . 1
        1.1.1. The Knowledge Builder application . . . 1
        1.1.2. Building blocks . . . 4
        1.1.3. Type hierarchy — Inheritance . . . 7
        1.1.4. Create and edit objects . . . 9
        1.1.5. Graph editor . . . 12
    1.2. Definition of schema / model . . . 24
        1.2.1. Define types . . . 24
        1.2.2. Relation types and attribute types . . . 33
        1.2.3. Model changes . . . 48
        1.2.4. Representation of schema in the graph editor . . . 53
        1.2.5. Metamodeling and advanced constructs . . . 56
        1.2.6. Indexing . . . 69
    1.3. Searches / Queries . . . 79
        1.3.1. Structured queries . . . 79
        1.3.2. Simple Search / Fulltext search . . . 99
        1.3.3. Search pipeline . . . 106
        1.3.4. Model "Hit" . . . 123
        1.3.5. Search in the Knowledge Builder . . . 125
        1.3.6. Special cases . . . 126
        1.3.7. Graph Query Language (GQL) . . . 130

1. Knowledge-Builder

1.1. Basics

When using i-views, databases work the way people think: simple, agile and flexible. That is why in i-views many things are different than relational databases: we do not work with tables and keys, but with objects and the relationships between them. Modelling of the data is visual and oriented towards examples so that we can also share it with users from the specialist departments.

With i-views we do not set-up pure data storage but intelligent Knowledge Graphs which already contain a lot of business logic and with which the behaviour of our application may, to a large extent, be defined. To this end we use inheritance, mechanisms for conclusions and for the definition of views, along with a multitude of search processes which i-views has to offer.

Our central tool is the knowledge builder, one of the core components of i-views. Using the knowledge builder we can:

- define the scheme but also establish examples and, above all, visualise
- define imports and mappings from a data source
- phrase requests, traverse graph data, process strings and calculate proximities
- define rights, triggers and views

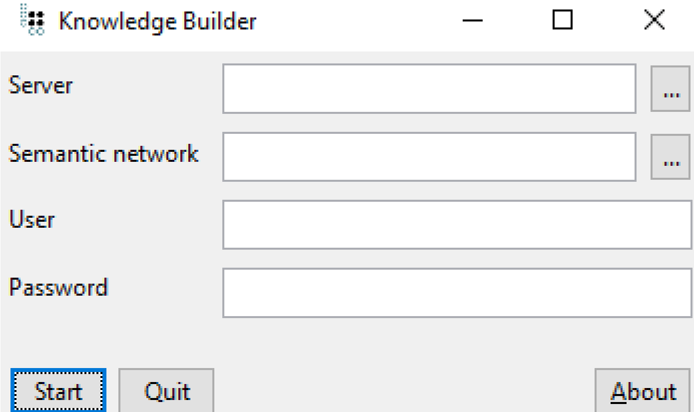
All these functions are the subject of this documentation.

1.1.1. The Knowledge Builder application

The executable application "kb" is an acronym for the i-views "Knowledge Builder" by which we administer the Knowledge Graph. When talking about the Knowledge Builder, we use special terms for orientation:

- **Backend:** The Knowledge Builder application (KB) by itself
- **Frontend:** Web frontend which is displayed in the browser by means of the viewconfiguration mapper application (VCM)
- **Volume:** The volume comprises all file data of the Knowledge Graph which is accessed by the Knowledge Builder.
- **Semantic element:** A semantic element is the smallest building block of the Knowledge Graph. An element can be either a type or an instance thereof, comprising object types and their objects, attribute types and their attribute instances as well as relation types and the individual relations.

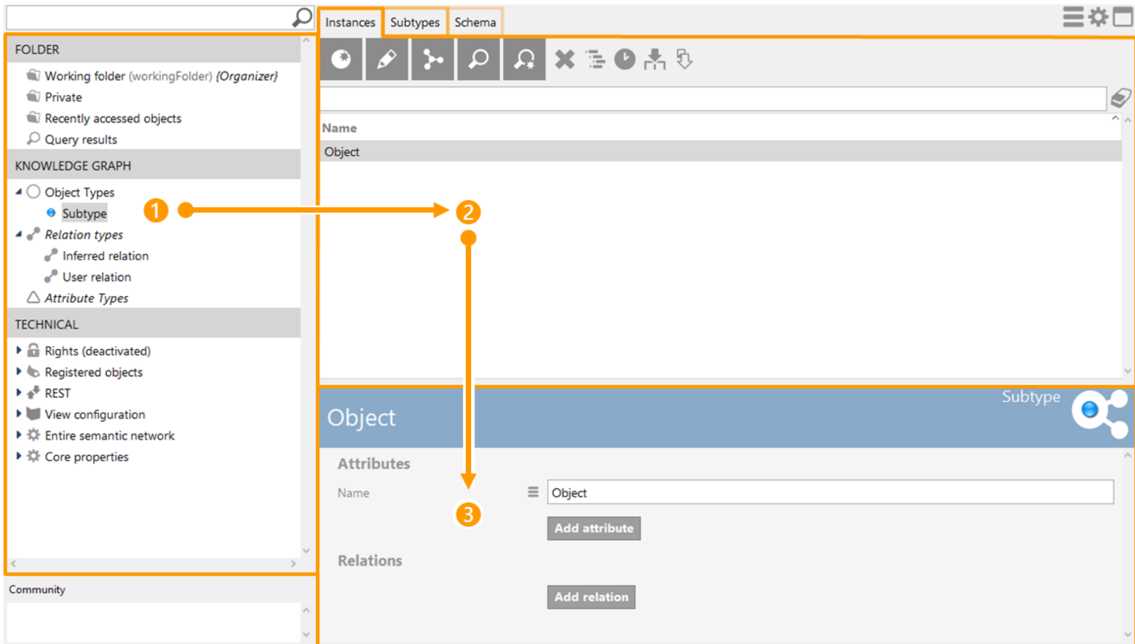
When we start the Knowledge Builder application, the login dialog is shown:



- **Server:** For the server, there are three kinds of server access available:
  - **(without server)** : The volume of the Knowledge Graph can be accessed via the local filesystem. In this case, the volume needs to be located within a "volumes" folder which is located in the same directory as the Knowledge Builder application itself. Since no mediator is in use, only one client application can access the volume at the same time — for example, the Knowledge Builder *or* the bridge for web frontend access.
  - **localhost** : This option is for accessing the volume via a mediator which is located int the same directory as the volumes folder and the Knowledge Builder. The mediator is an additional application that allows simultaneous access of different client applications, for example Knowledge Builder *and* bridge for web frontend access.
  - **Server address and server port** : Since the Knowledge Builder is preferably used as one of many clients that grants collaborative access to the Knowledge Graph volume on a server via a mediator, this is the most often used kind of access. Server address an port are written colon-separated in forms of *serveraddress:portnumber*.
  - **Knowledge Graph:** The name of the relevant existing volume must be specified here. **
- **User:** User name for volume access.
- **Password:** Password for volume access.

NOTE	For creating a new volume, the admin tool is needed.

The Knowledge Builder user interface is divided into following areas:



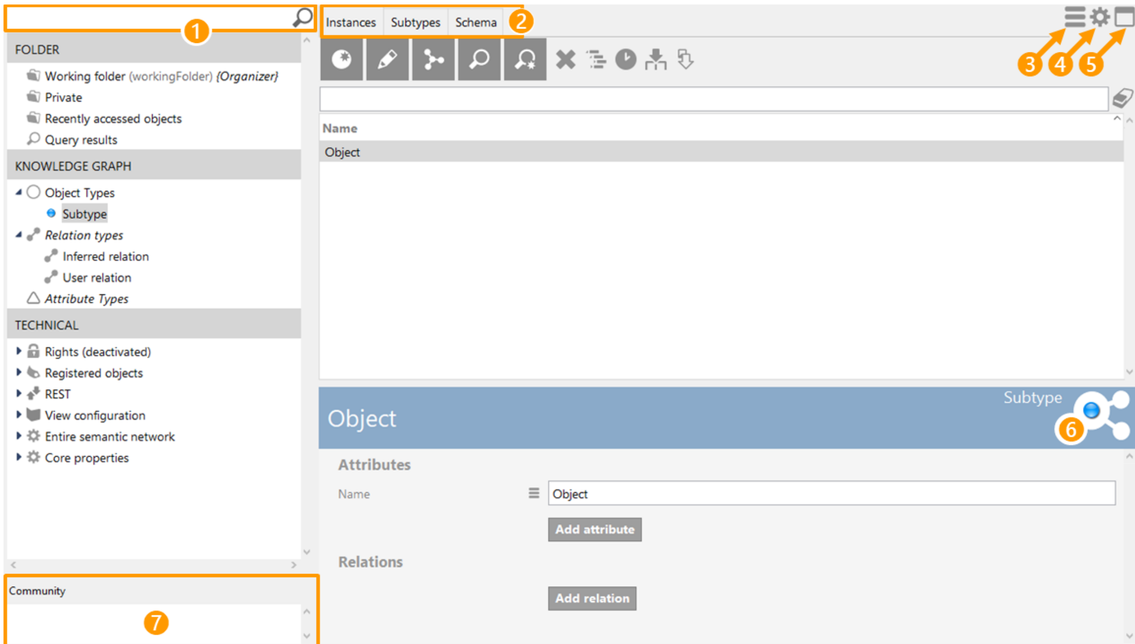
* ❶ **Organizer:** Type hierarchy view on the left side of the Knowledge Builder screen.

* ❷ **Instance list/object list/list view:** Upper right part of the Knowledge Builder that shows the instances of the respective type which has been selected in the organizer. Instance lists only contain table views. If severeal table views are defined for one type, they are separated by tabs.

- ❸ **Detail editor / detail view:** Lower right part of the Knowledge Builder in which a detailed view of the instance is shown which has been selected in the instance list. The detail view is able to contain several type of views.

Therefore, editing properties of a semantic element is done by first selecting the subtype in the organizer ❶, then selecting the instance of the list view ❷ and by editing the properties in the detail editor ❸.

Besides the areas, there are further actions and selections available as follows:



- ❶ **Global search:** The global search works for all elements of the Knowledge Graph. Additional searches can be added via drag&drop of queries from the folders into the search input field.
- ❷ **List tabs:** The list views are divide up into instance list and subtypes list. As a new feature since i-views 5.4, a schema tab provides a sole detail editor for schema definition of properties and property types for the selected subtype.
- ❸ **Global actions:** The main menu of the Knowledge Builder offers element-independent actions for the user. For more information, see the respective chapter at the beginning of the i-views Knowledge Builder Technical Handbook.
- ❹ **Global settings:** The global settings provide user dependent settings for every user and administrative settings which are available for administrators only. For more information, see the respective chapter at the beginning of the i-views Knowledge Builder Technical Handbook.
- ❺ **New window:** This button allows opening listed views, such as import mappings etc. so that the window keeps persistent despite a different selection in the organizer.
- ❻ **Context menu:** This context menu provides all actions concerning the relevant semantic element. Clicking onto the big circle opens the context menu, clicking on one of the small circles opens the element in a graph editor. The big circle is also for dragging and dropping the element into the graph editor or a semantic elements folder.
- ❼ **Community:** If several users are logged in, the are listed here and can be contacted via chat for collaborative work.

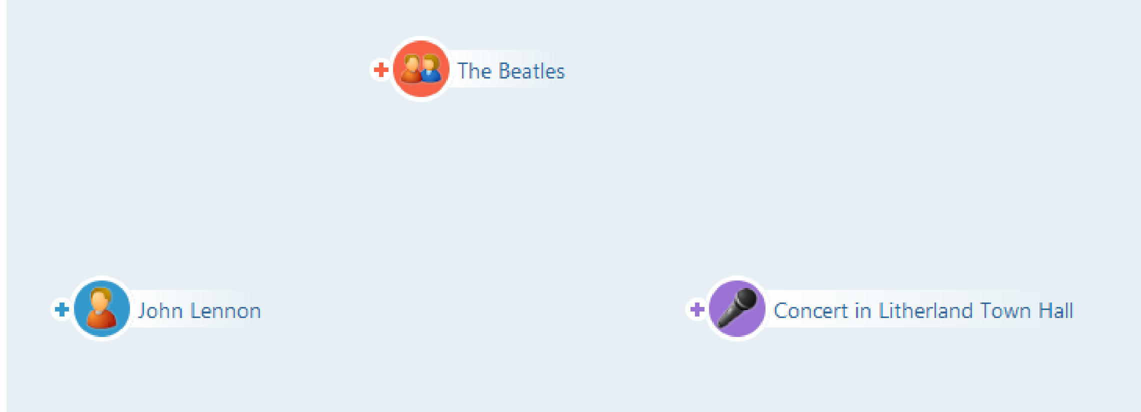
For further information, see the following chapters.

1.1.2. Building blocks

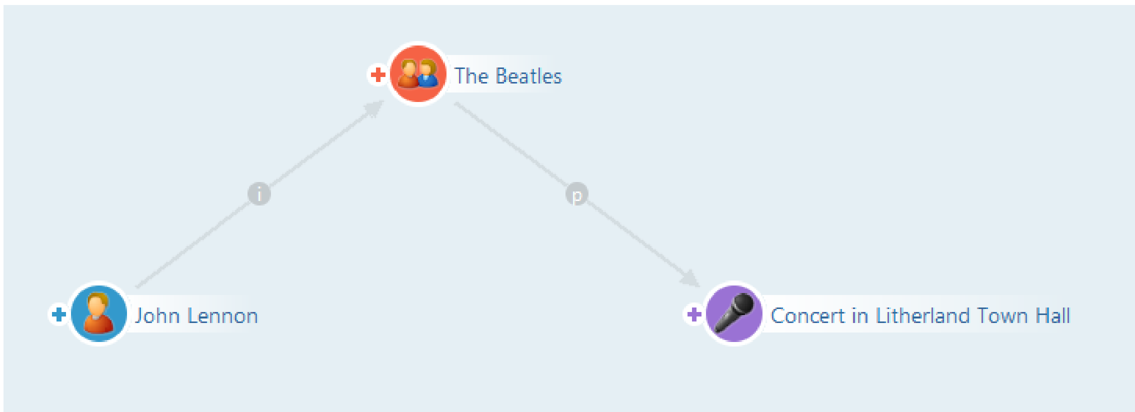
The basic components of modelling within i-views are instances and their types:

- objects
- relations
- attributes
- object types
- relation types
- attribute types

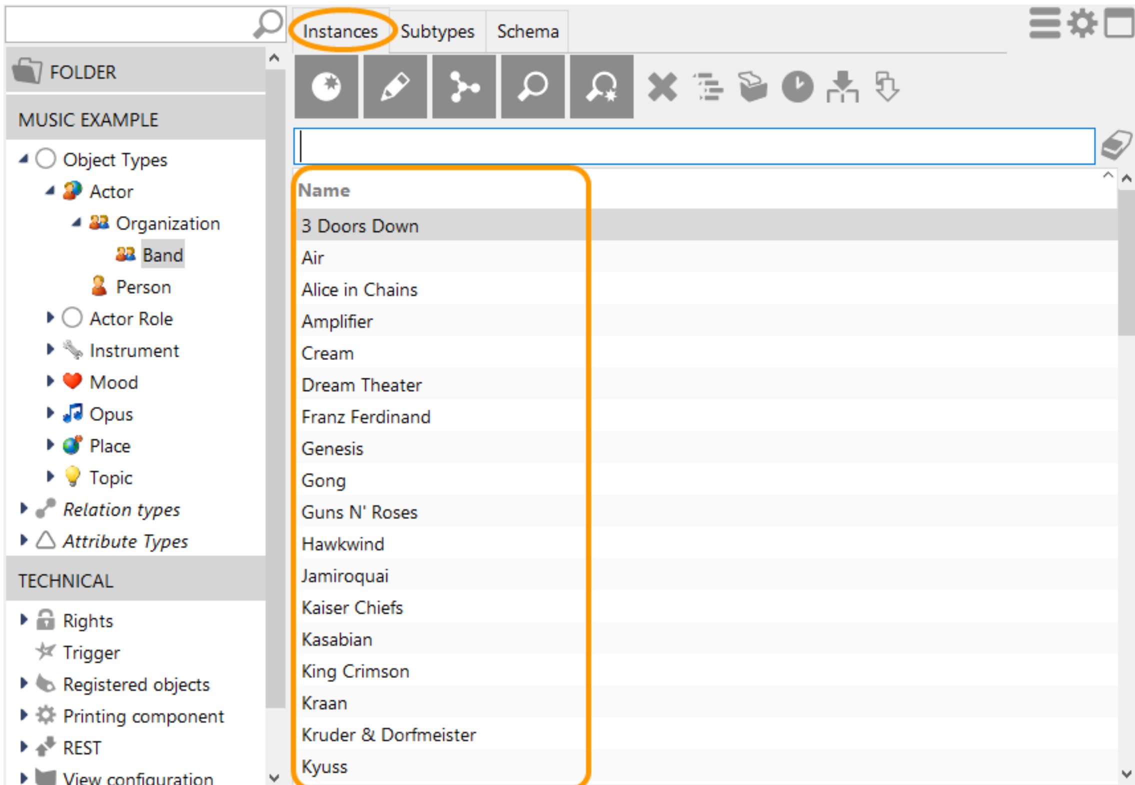
Examples for specific objects are John Lennon, the Beatles, Liverpool, the concert in Litherland Town Hall, the football world cup in Mexico in 1970, the leaning tower of Pisa, etc.:



We can link these specific objects together through relationships: "John Lennon is a member of the Beatles", "The Beatles perform a concert in Litherland Town Hall".

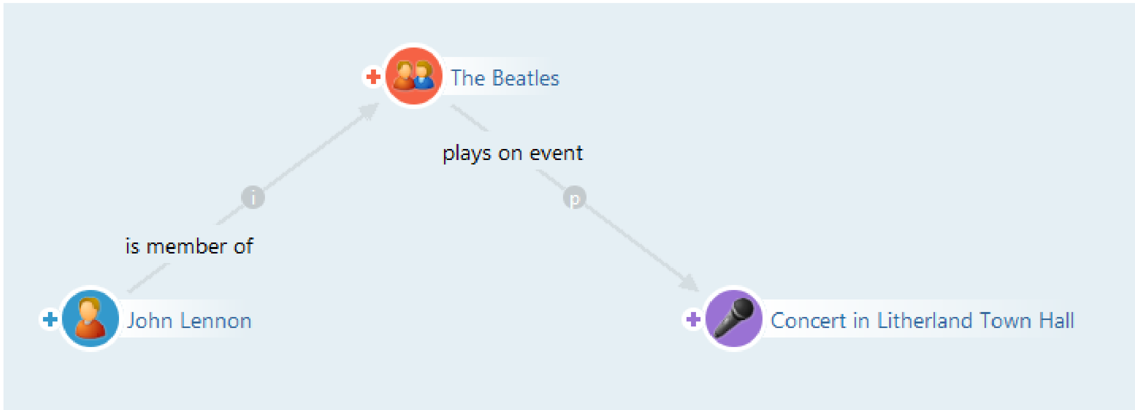


Additionally, we have introduced four types here: specific objects always have a type, e.g. the type of persons, type of the cities, the events or the bands — types which you may freely define in your data model.



The main window of i-views: on the left-hand side the types of objects, on the right-hand side the respective, specific objects — here we can also see that the types of the i-views Knowledge Graphs are within a hierarchy. You will find out more about the type of hierarchy in the next paragraph.

Even the relationships have different types: between John Lennon and the Beatles there is the relationship "is member of"; between the Beatles and their concert the relationship could be called "performed at" - if we want to generalise more, "participates in" is perhaps a more practical type of relationship.

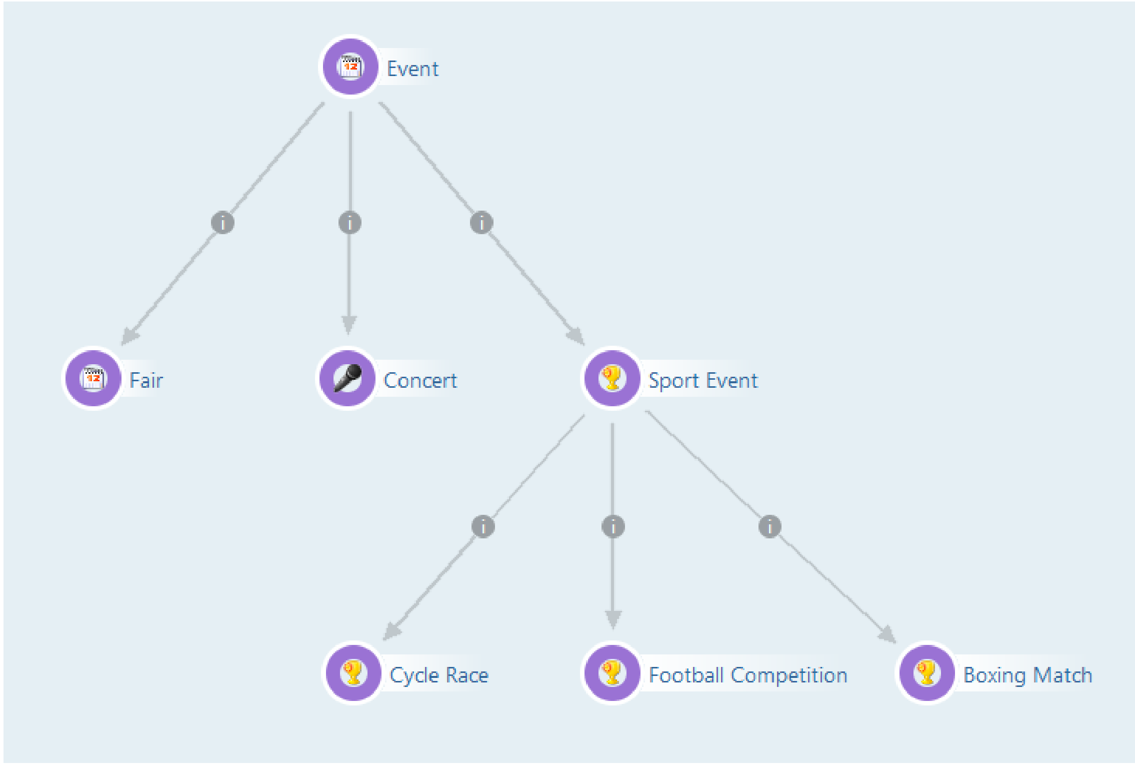


The same applies for attributes: in the case of a person these may be the name or the date of birth. Specific persons (objects of the type "person") may then have name, date of birth, place of birth, address, colour of eyes, etc. Events may have a location and a time span. Attributes and relations are always defined with the object itself.

1.1.3. Type hierarchy — Inheritance

We can finely or less finely divide types of objects: we can put the football world cup in 1970 into the same basket as all the other events (the book fair in 2015, the Woodstock festival, etc.), then we only have one type called "event" or we differentiate between sport events, fairs, exhibitions, music events, etc. Of course, we can divide all these types of events even finer: sport events may, for example, be differentiated by the types of sports (a football match, a basket ball match, a bike race, a boxing match).

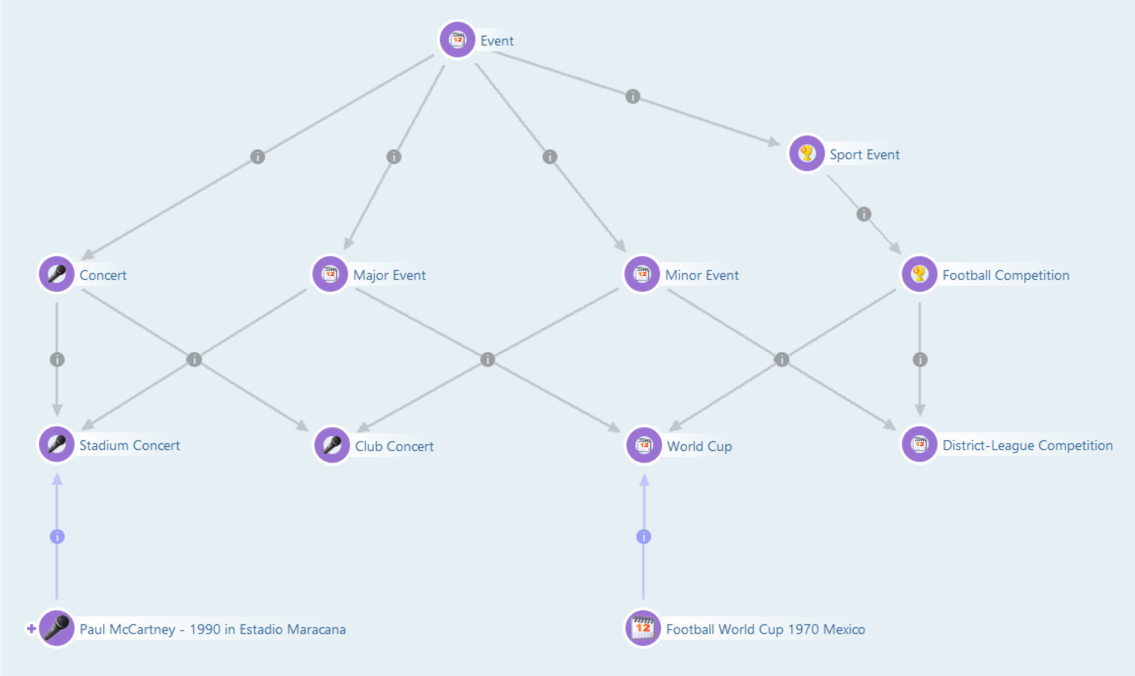
In this manner we obtain a hierarchy of supertypes and subtypes:



The hierarchy is transitive: when we ask i-views about all events, not only all specific objects are shown which are of type event, but also all sports events and all bike races, boxing matches and football matches. Hence, since the type "boxing match" is not only a subtype of "sport event", i-views will reject a direct supertype / subtype relationship between event and boxing match — with a note that this connection is already known.

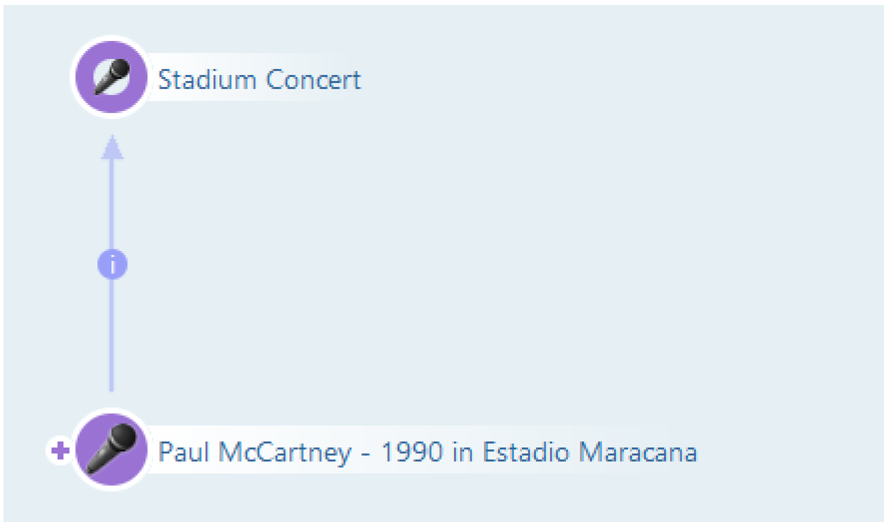
The hierarchical structure does not necessarily have to have the structure of a tree — a type of object may also have several upper types. However, an object may only have one type of object.

If we then wish to join the aspects of a concert and major event we cannot do this in the specific concert with Paul McCartney because we need the type of object "stadium concert" in order to do this:

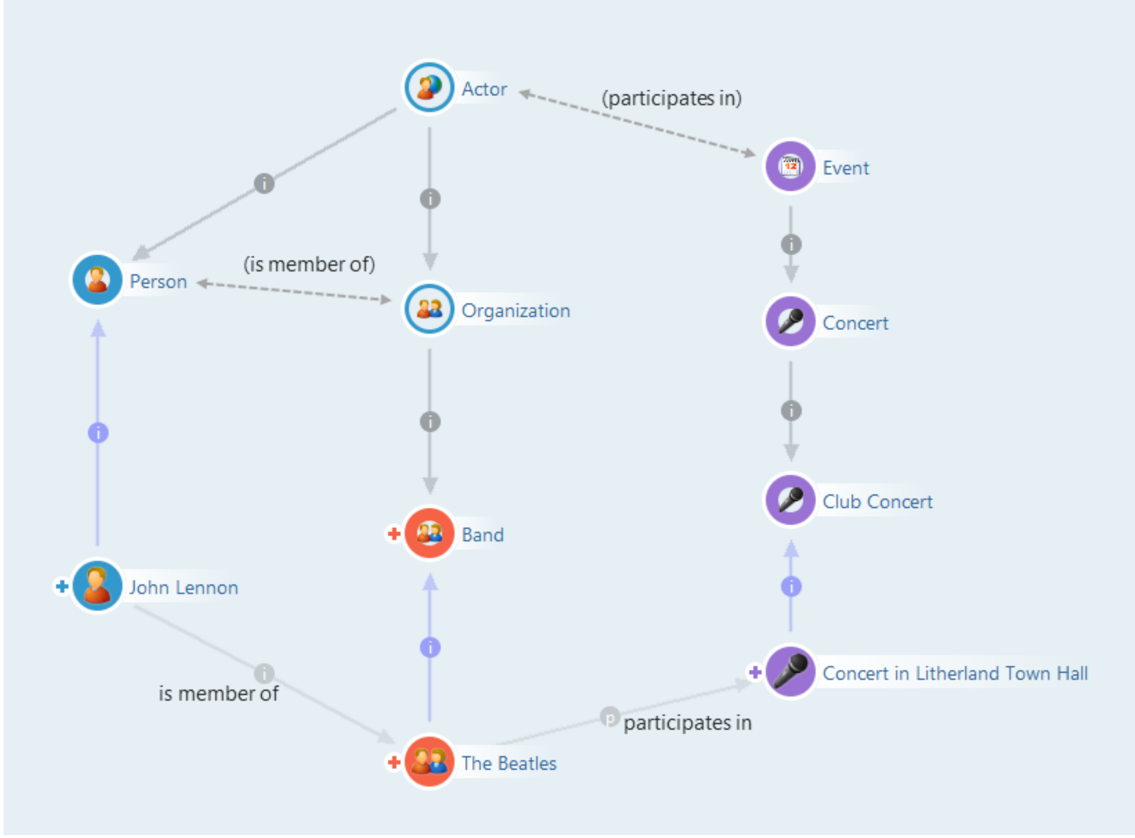


Type hierarchy with multiple inheritance

The affiliation of specific objects with a type of object is also expressed as a relation in i-views and may as such be queried:



When do we differentiate between types at all? Types do not only differ in icon and colour — their properties are also defined in the types and when queried, the types can also easily be filtered. The inheritance plays a major role in all these questions: properties are inherited, icons and colours are inherited and when, in a query, we say that we wish to see events, all objects of the subtypes are also shown in the results.

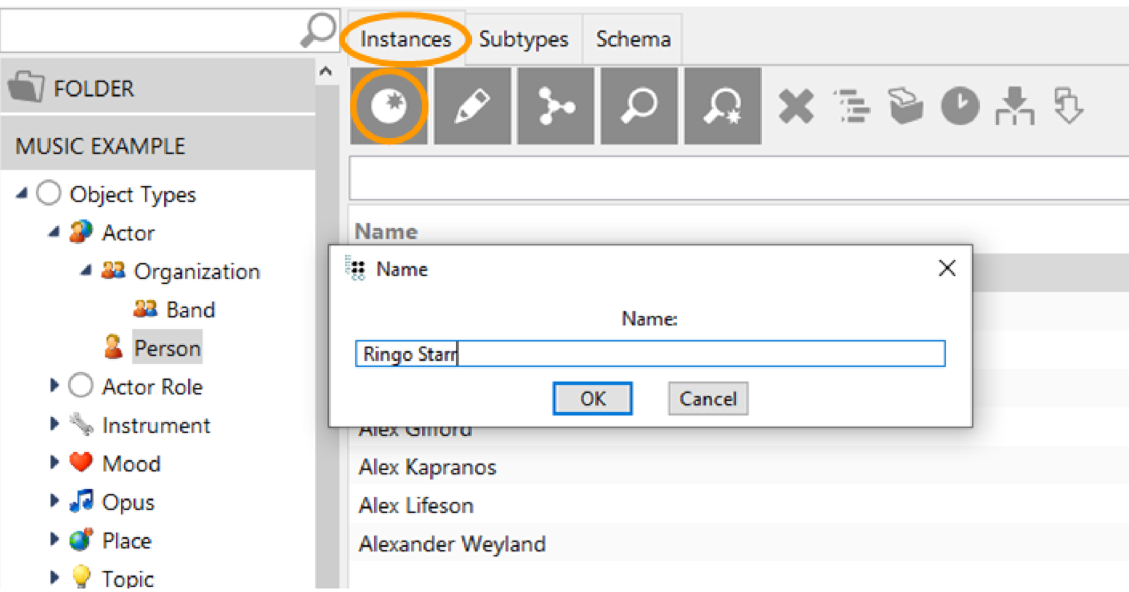


Inheritance makes it possible to define types of relations (and types of attributes) further up in the hierarchy of the object type and hence use them for different types of objects (e.g. for bands and other organisations.

1.1.4. Create and edit objects

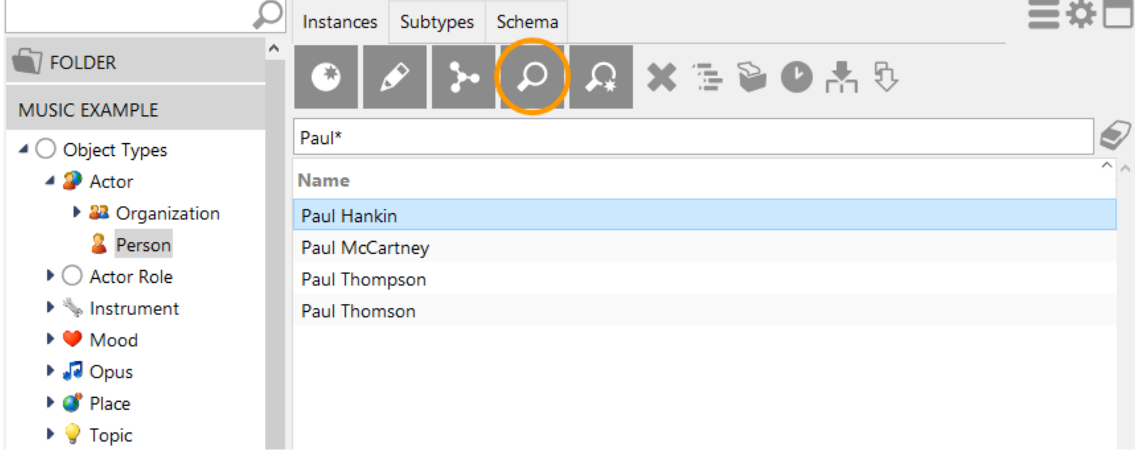
Creating specific objects

Specific objects (in the knowledge builder they are called "instances") may be created everywhere within the knowledge builder where types of objects can be seen. Based on the types of objects, objects can be newly created via the context menus.



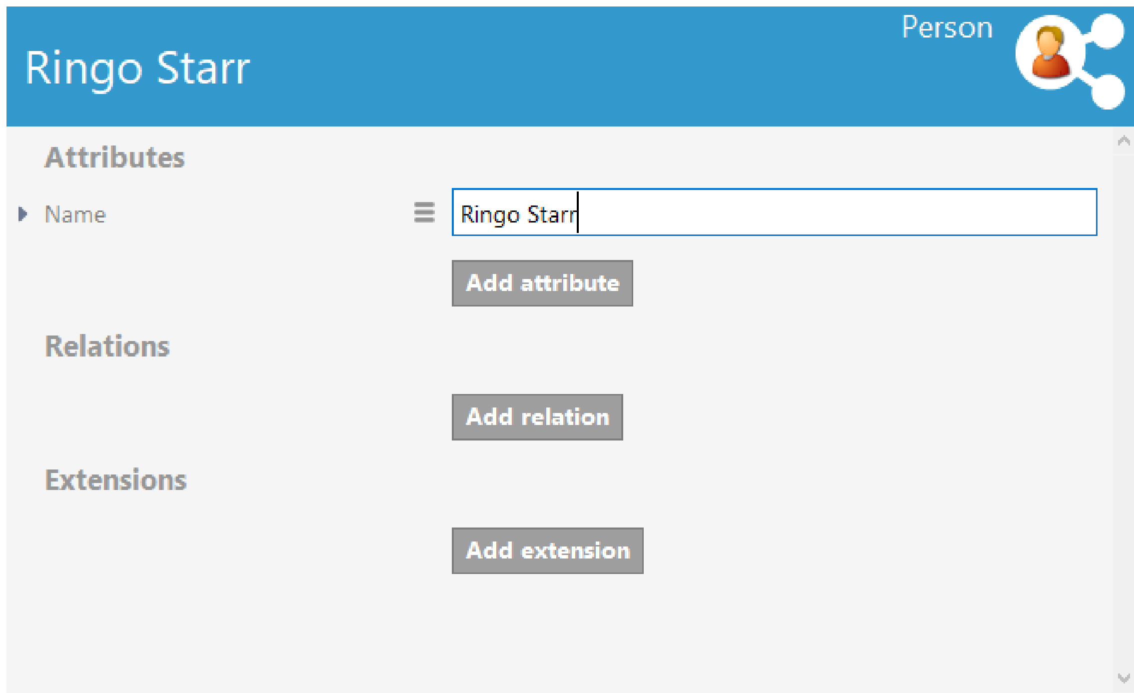
An object can be created by means of the button "new" and using the named entered

In the main window below the header there is the list of specific objects already available. In order that objects cannot inadvertently be created twice, the name of the object can be keyed into the search button in the header. The search does not, by default, differentiate between upper and lower case and the search term may be cut off left and right (supplement by placeholders "*" and "?"):



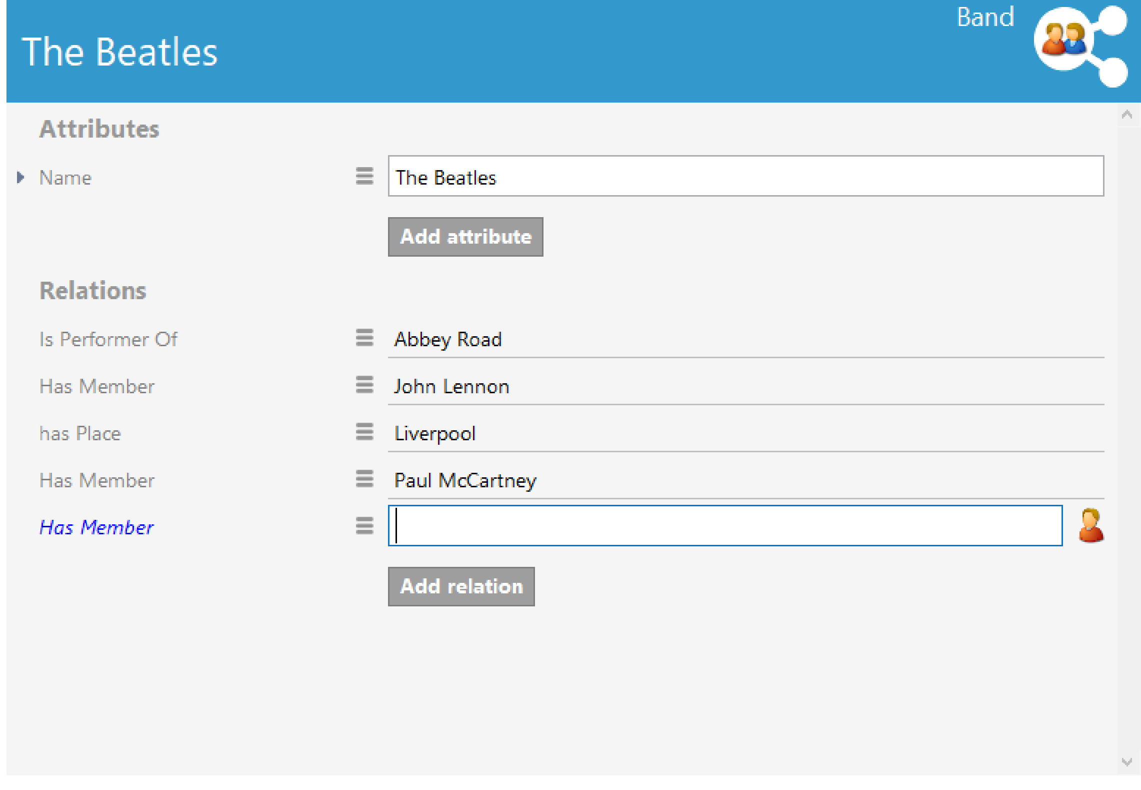
Editing objects

After entering and confirming the name of the object, further details for the object created may be keyed into the editor. The object may be assigned attributes, relations and extensions by using the respective buttons.

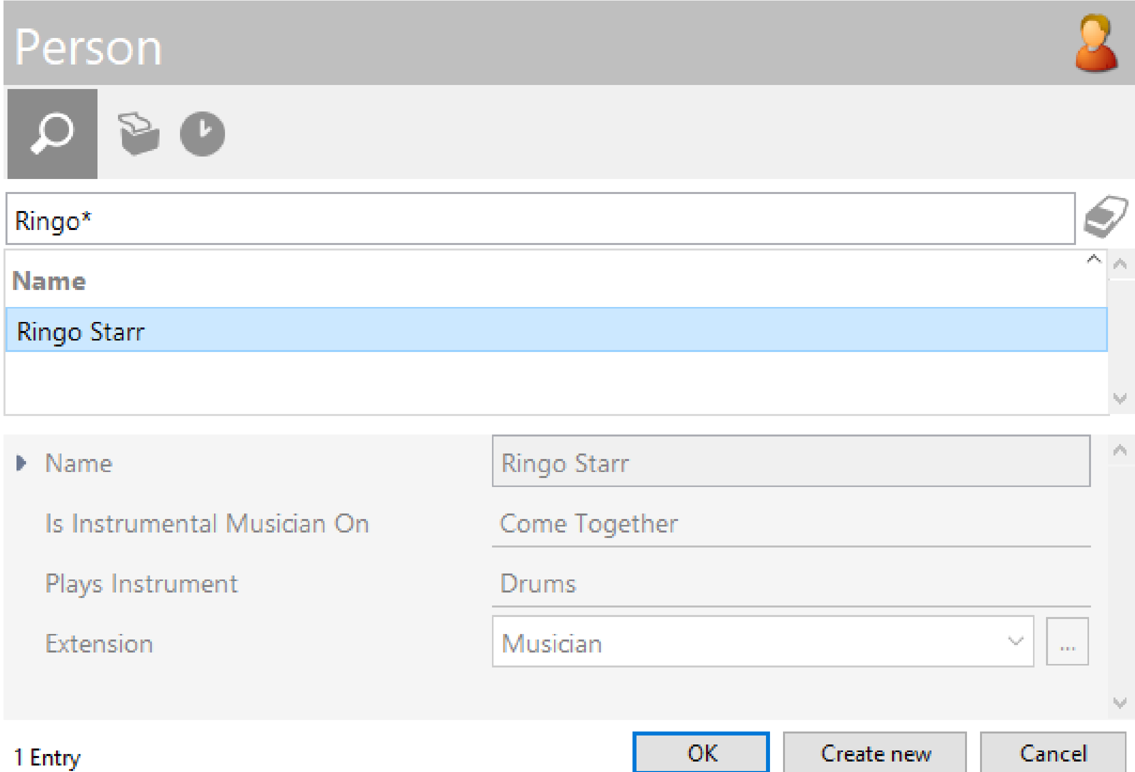


When editing an object we can, in addition to linking it to another object, also generate the target of the link if the object does not already exist.

For example, members of a music band are documented completely. Via the relation, we want to link the member Ringo Starr with the object "The Beatles". If it is not yet clear whether the object Ringo Starr is already documented in i-views you can use the search button to ascertain this,



or via the icon button, select "Choose relation target" from a searchable list with all feasible targets of relation.



Deleting the relation has a member may be accomplished in two different ways:

1. Delete in the context menu using the button *further actions* and the option "*delete*".
2. With the cursor over the button *further actions* and holding down the Ctrl key.

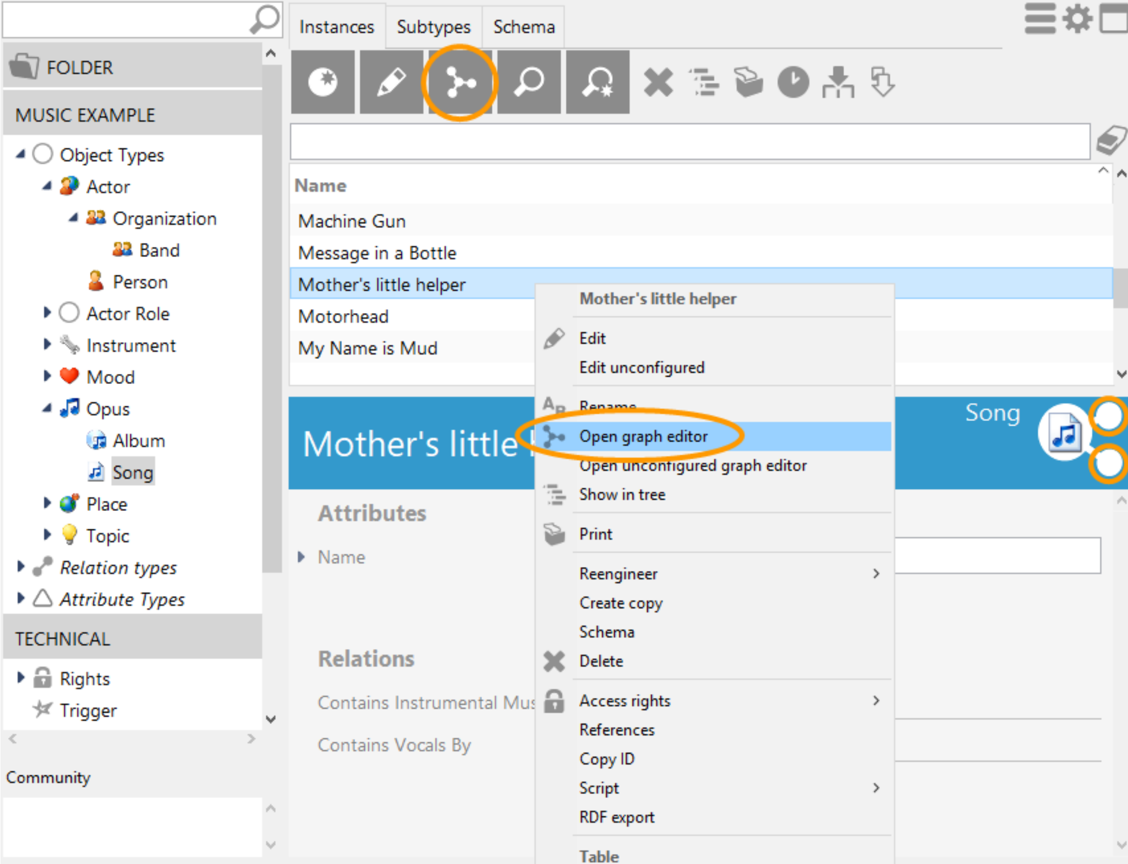
The target object of the relation itself will not be deleted as a result of this however. If an object has to be deleted this is done via the button in the main window or via the context menu directly on this object.

Objects may also be created using the graph editor. This process is described in the following paragraphs.

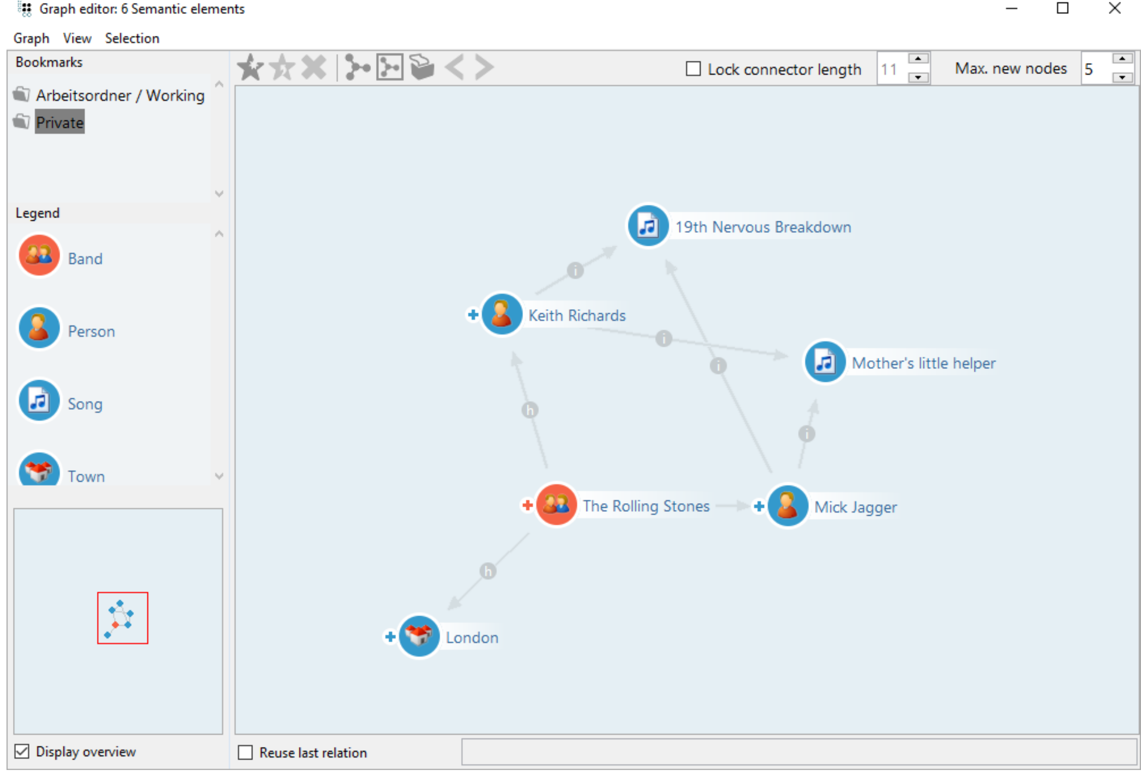
1.1.5. Graph editor

1.1.5.1. Introduction graph editor

By using the graph editor, the Knowledge Graph with its objects and links can be depicted graphically. The graph editor may be opened on a selected object using the *Graph* button.



Objects will be displayed as nodes and the relations between them as arrows.



The graph always shows a section of the Knowledge Graph. Objects from the graph may be displayed and hidden and you can navigate through the graph.

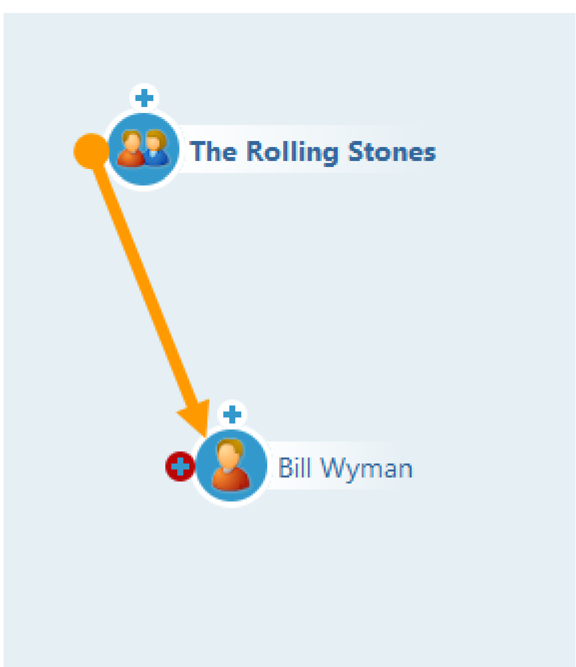
Scrolling on the Graph will zoom at the position of the cursor. The displayed section can be moved by clicking and dragging the background or by using the arrow-keys.

In the bottom left corner is an overview of the whole Graph which shows all nodes, edges and what section of the Graph is currently visible. Clicking on the overview will move the right window to the clicked-on section. Scrolling in the overview will zoom at the position of the cursor. Below the overview is the option to disable it.

On the left-hand side of a node there is a drag point for interaction with the object. By double-clicking on the drag point all user relations of the object will be displayed or hidden.

Linking objects via a relation is carried out in the graph editor as follows:

1. Position the cursor over the drag point to the left of the object with the left mouse button.
2. Drag the cursor in a held down position to another object (drag & drop). If several relations are available for selection, a list will appear with all feasible relations. If there is only one feasible relation between the two objects, this will be selected and no list will be shown. An already existing relation can be reassigned to another element by drag & drop, if the schema definition allows this.



Re-use last relation

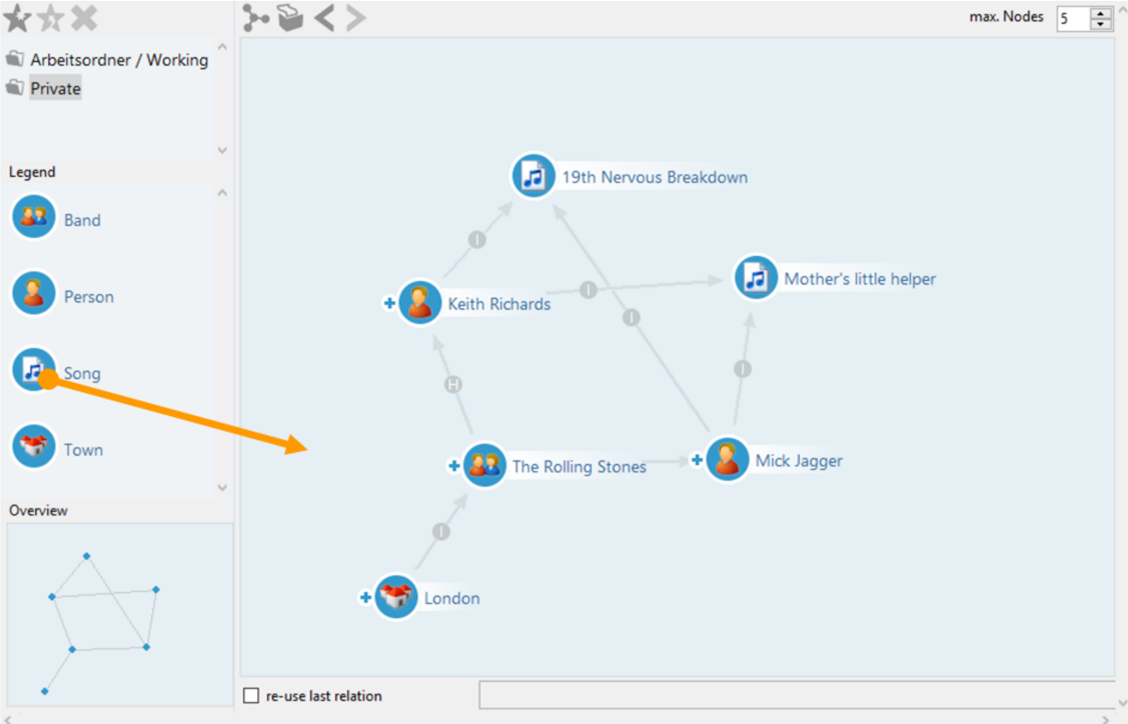
If this box is checked, the user will not be asked which relation they want to create when drawing a new one between two nodes. Instead, the new relation will be the same as the last one that was drawn. The locked relation is shown in the box to the right.

In order to display objects in the graph editor there are different options:

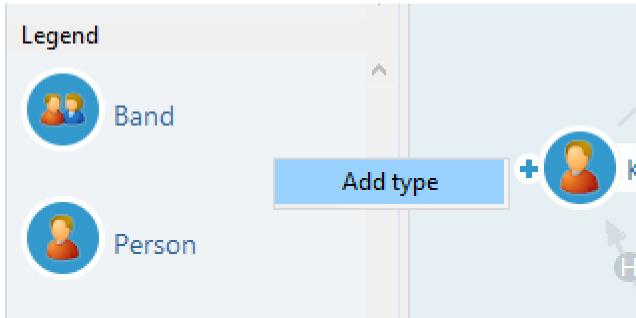
- Objects may be dragged from the hit list in the main window to the graph editor window using drag & drop.
- If the name of the object is known it can be selected via the context menu using the function "show individual".

NOTE	If an object is to be hidden from the graph editor, it may be removed from there by clicking it and dragging it from the graph editor holding down the Ctrl key. In doing so, there will be no changes in the data: the object will exist unchanged within the Knowledge Graph, but it will not be displayed anymore in the current graph editor section.

New objects may also be created in the graph editor. To do this we drag & drop the type of object from the legend on the left-hand side of the graph editor to the drawing area:



If there are no types of objects to be seen in the legend you can search for them using a right mouse click in the legend area. Following this, the name of the object will be given.



The editor will re-appear in which the possible relations, attributes and enhancements for the object can be edited.

1.1.5.2. Operations on objects in the graph editor

The context menu of nodes consists of multiple parts:

- The title, which is the name of the object. This has the normal context menu of the object as its submenu.
- Operations that effect the object itself. These are some operations from the normal context menu for ease of access. It additionally contains the option to merge two objects.
- Operations that effect the graph editor and its nodes, without changing any objects:

  Fix
  : Fixed nodes will not be moved by the layout function.

  Unpin
  : Reverses the fixing of nodes.

  Shortest path
  : Adds nodes for all objects that represent the shortest connection between two selected nodes. If only one node is selected, a dialog will ask which node it should be connected to.

  Center node
  : The selected node will be moved to the center of the displayed area.

  Show
  : These are operations to add new nodes to the graph depending on the selected node:

    Attributes
    : Adds all attributes defined at this object.

    Instances
    : Adds all instances of this type.

    Extension types
    : Adds all extension types definied for this type.

    Inferred relations
    : Adds all objects of the selected inferred relations.

    Property types
    : Adds all property typs defined for this type.

Property instances

Adds all concrete attributes of this type, which are defined at objects in this graph.

Defined for

Adds all types for which this property type is defined.

Extensions

Adds all extensions that are defined at this object.

Hide

These are operations to remove nodes from the graph depending on the selected node:

Selected nodes

Removes all selected nodes.

Related nodes

Removes all nodes that are connected to this node.

Subtypes

Removes all subtypes of this type.

Instances

Removes all instances of this type.

Property instances

Removes all concrete attributes of this attribute type.

Display

These operations change the appearance of nodes:

Small icons

Changes the node size to small.

Medium sized icons

Changes the node size to medium.

Large icons

Changes the node size to large.

New label

Changes the label of the node.

Change icon

Changes the icon of the node to an icon that is defined at itself, its type or a super type.

NOTE	It is also possible to select multiple objects at the same time by holding down the shift key while clicking on the background and dragging the mouse, or clicking the desired nodes. All selected objects can be moved together (for example by pressing Ctrl + arrow keys), and operations are carried out on all selected objects simultaneously.

1.1.5.3. View

The menu **View** provides many more functions for the graphic illustration of objects and types of objects:

Default settings

Opens the KB settings with the graph editor settings. These are also available in the KB setting window ⚙ under **Personal** > **Graph**.

Change Background

The background color can be changed or a picture can be set as background.

Auto hide nodes

Automatically hides surplus nodes as soon as the number of desired nodes is exceeded and shown. The number can be set in the input field "max. new nodes" in the toolbar.

Auto layout nodes

Automatically implements the layout function for newly displayed nodes.

Node alignment

If enabled, nodes will snap to the x or y coordinates of close nodes when moving them.

Fix all labels

Using this option the names of all relations are always visible, not only when rolled over with the mouse. Alternatively, the description may be fixed directly in the context menu of a relation.

Show internal names

Displays the internal name of types of in brackets.

Recover hidden edges

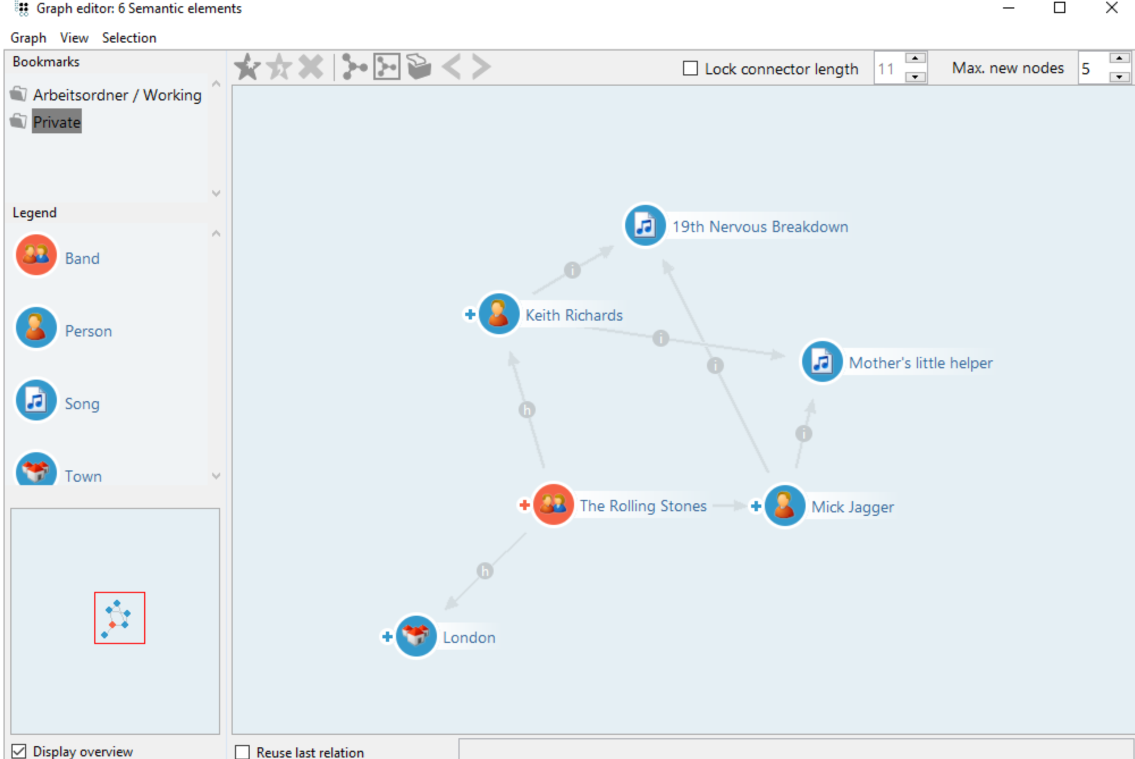
All edges hidden by means of the context menu are shown again.

Always highlight edges

All edges will be highlighted at all times.

The window of the graph editor and the main window of the knowledge builder provide even more menu items which may offer support when modelling the Knowledge Graph.

On the left-hand side of the graph editor window there is the legend of the types of objects.



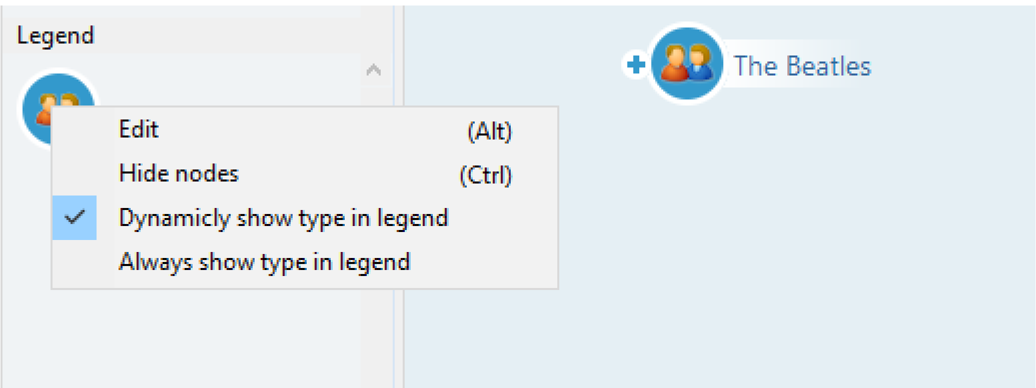
This legend shows the types of objects for the specific objects on the right-hand side.

Via drag&drop of an entry from the legend into the drawing area you can add or create a new specific object of the corresponding type.

When right-clicking into the legend area, further types can be added permanently to the legend so that objects of that type can be added to the graph by means of drag & drop.

NOTE You can drag & drop elements from the Knowledge-Builder into the graph editor when holding down the Ctrl key.

Via the context menu for the legend entries all specific objects can be hidden from the image. Here you can also "hold" legend entries and add new types of objects to the legend (regardless of whether specific objects of this kind are represented in the image).

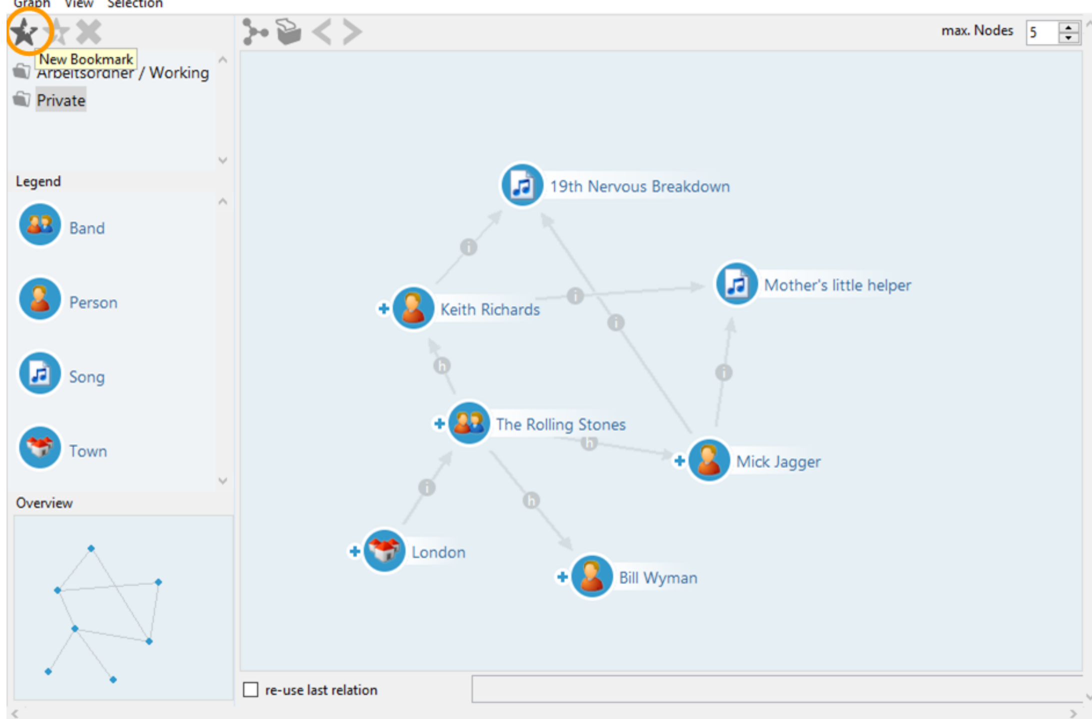


1.1.5.4. Bookmarks and further controls

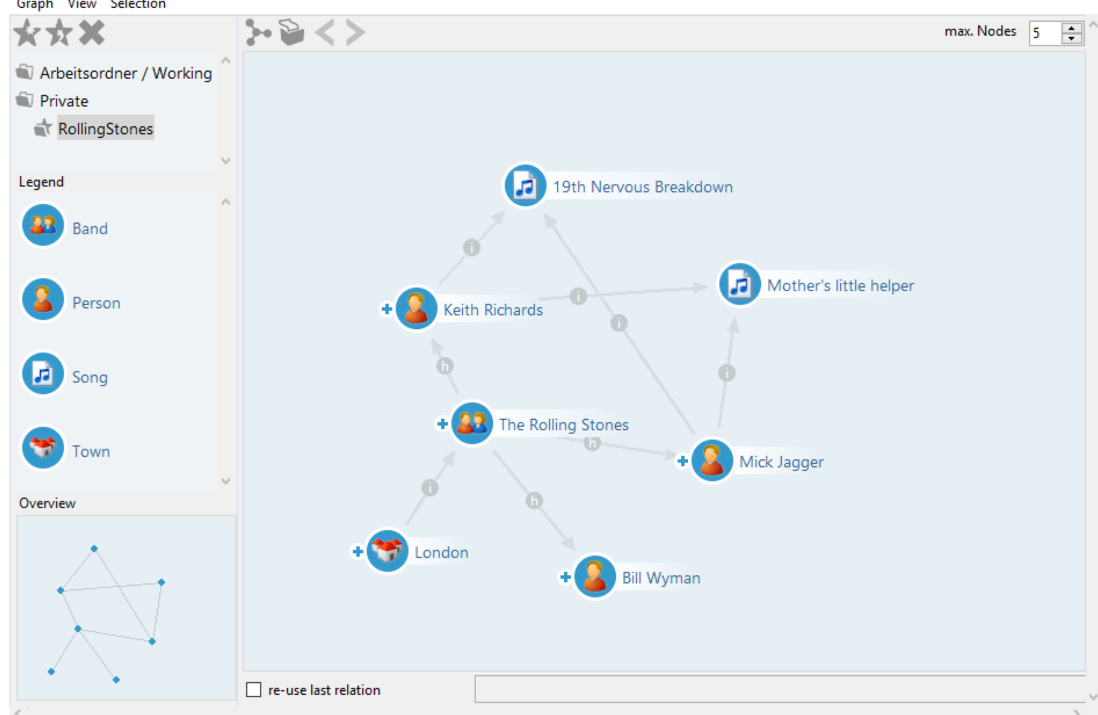
The toolbar contains some useful controls:

Bookmarks

Parts of the Knowledge Graph or "subgraphs" can be saved as bookmarks. The objects are saved in the same position as they are placed in the graph editor.



When a bookmark is created it may be given a name. All nodes contained in the bookmark are listed in the description of the bookmark.



Bookmarks are no data backups, though: objects and relations which were deleted after a bookmark was saved are also no longer available when the bookmark is shown.

Layout

The layout function enables you to position nodes automatically within the display area at the currently selected zoom level when many nodes are not allowed to be positioned manually. When more nodes are displayed they will also be automatically positioned in the graph via the layout function. The option "auto layout nodes" must be activated for this purpose (see previous chapter).

Adjust view to include all nodes

In contrast to the layout function this will adjust the displayed area to show all nodes.

Print

Opens the dialogue window for printing or for generating a PDF file from the displayed graph.

History

Using the buttons "reverse navigation" and "restore navigation", elements of a (section of) a Knowledge Graph may be hidden again in the order of sequence in which they were shown (and vice versa). Furthermore, these buttons reverse the auto layout. The buttons can be found in the header of the graph editor window or in the menu "graph".

Lock connector length

This locks the distance at which new nodes are added. If the box is unchecked the distance is relative to the current zoom-level.

Max. new nodes

If a node / an object has many adjacent objects, it often doesn't make much sense to display all of them when clicking on the drag point. For this reason, the maximum amount of nodes to be displayed at once can be set.

1. Via the global settings in the tab "Personal", the maximum amount of new nodes can be set.
2. Within the graph editor, the amount can also be set in the upper right corner. max. Nodes 5

If the drag point has been clicked to show the adjacent objects a selection list will appear instead of the objects.

The menu **Graph** contains more functions for the graph editor:

Copy into the clipboard

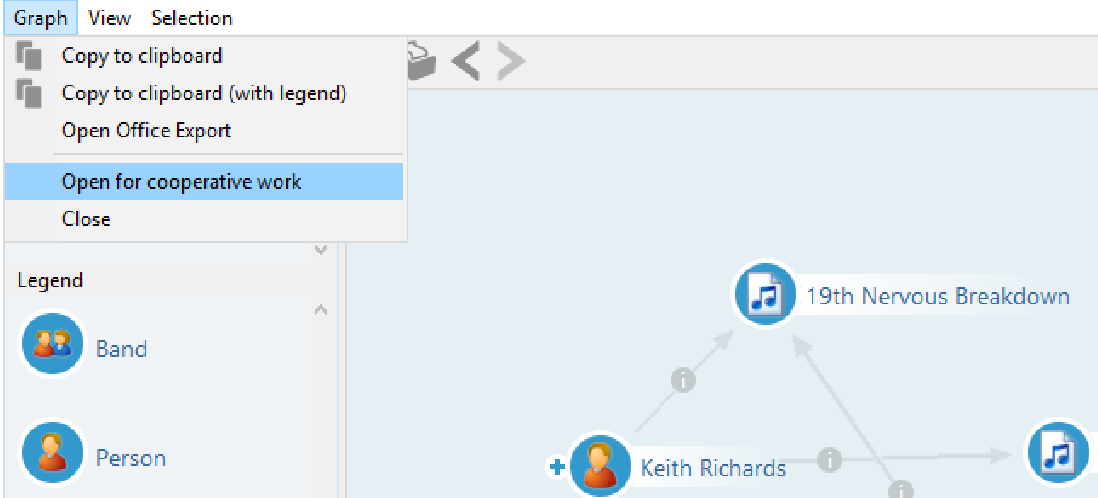
This function creates a screenshot of the current contents of the graph editor. This image may then be inserted into a drawing or picture processing programm, for example.

Copy into clipboard (with legend)

Same as the previous option, but the picture will include the legend.

Open for cooperative work

This function enables other users to work on the graph mutually and simultaneously. All changes and selections of a user on the graph (layout, showing/hiding nodes, etc.) will then be shown to all other users synchronously.



1.1.5.5. Settings

The settings for the graph editor can be found in the KB settings under **Personal** > **Graph**. They can also be opened via graph editor using the menu **View** > **Default settings**.

Some of these settings can be overridden in the graph editor. This will only apply to that specific

graph editor window.

Show tooltips with details

If true, hovering a node will display a tooltip listing the objects properties.

Max. number of lines for tooltips

Defines at what point the tooltip will be cut off.

Auto hide nodes

Automatically hides surplus nodes as soon as the number of desired nodes is exceeded and shown. The number can be set in the input field "max. new nodes" in the toolbar.

Auto layout nodes

Automatically implements the layout function for newly displayed nodes.

Node alignment

If enabled, nodes will snap to the x or y coordinates of close nodes when moving them.

Use Cairo library to display the graph

Can only be activated if the cairo library can be found by the KB executable. If true, the library will be used to display certain things.

Default zoom

The zoom level at which a graph editpr will be openend.

Max. new nodes

If a node / an object has many adjacent objects, it often doesn’t make much sense to display all of them when clicking on the drag point. This defines how many new nodes can be added at once without being prompted to choose specific nodes.

Max. label length

Defines after how many symbols the node labels will be cut off.

Node size

Defines at which size a node is added to a graoh.

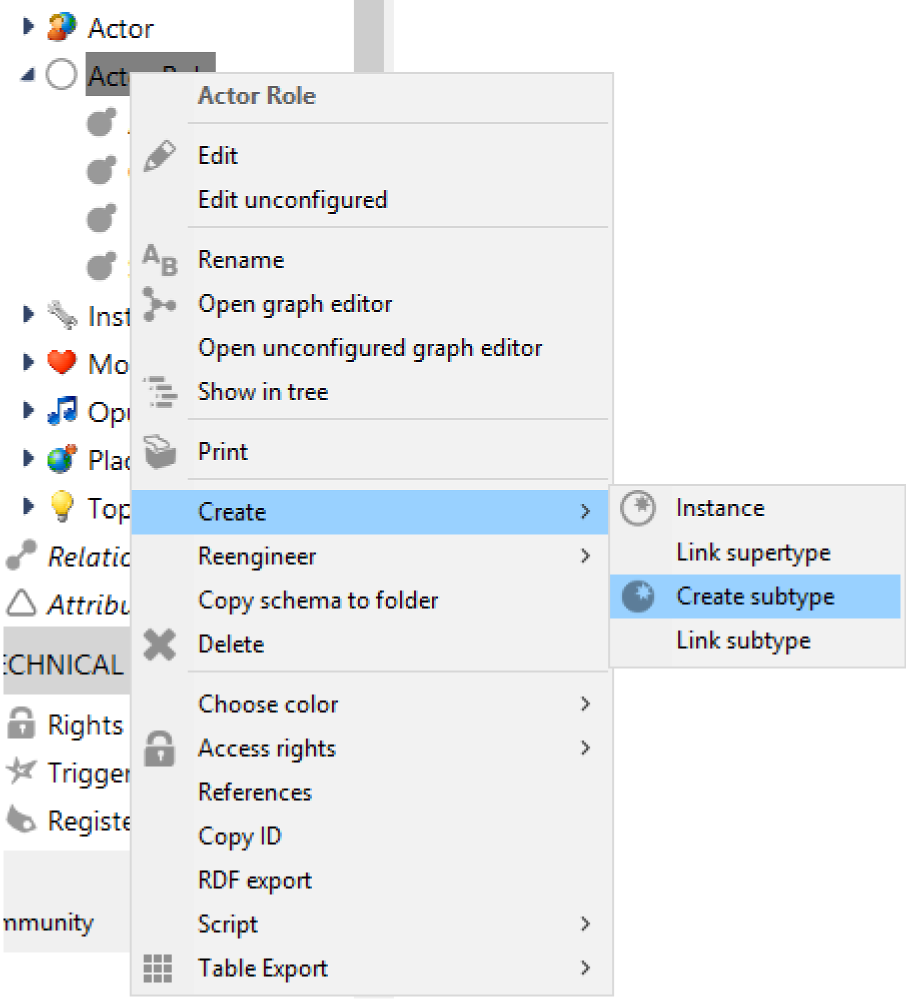
Legend configuration

Here you can define types, which will always be displayed in the legend when opening a graph editor.

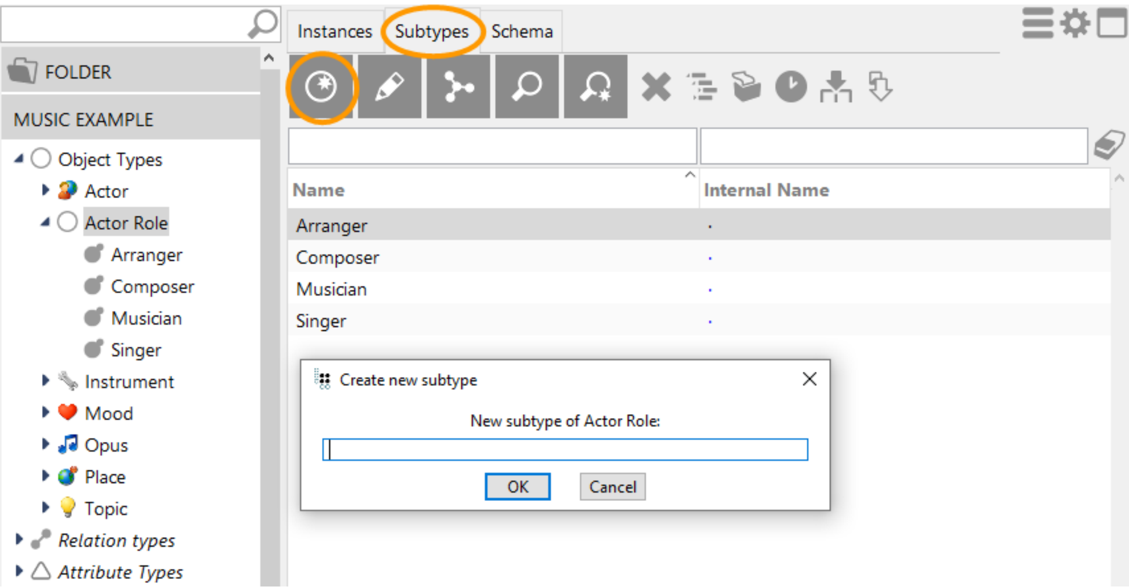
1.2. Definition of schema / model

1.2.1. Define types

The principle of the type hierarchy was already presented in chapter Building blocks. If new types are to be created this is always done as a subtype of a type which already exists. Creating subtypes can be carried out either via the context menu Create > Subtype



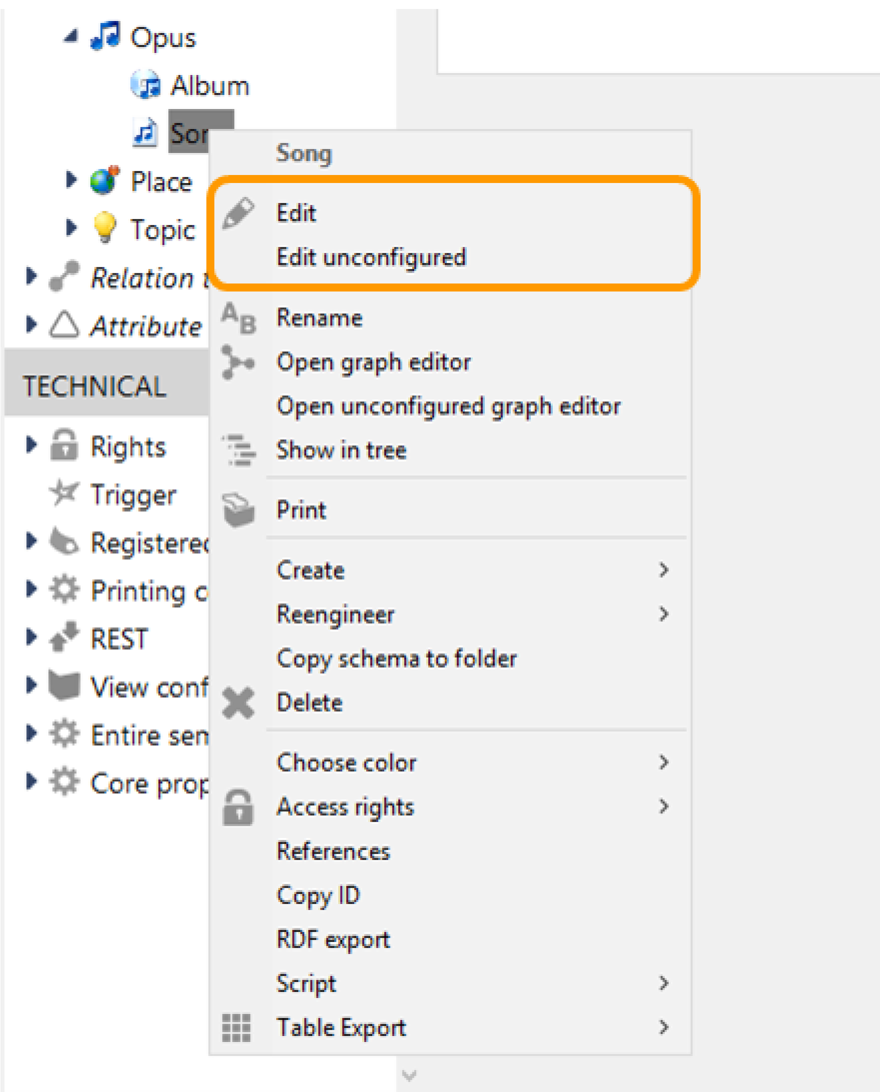
or in the main window using the tab "Subtypes" above the search field and the tab "new":



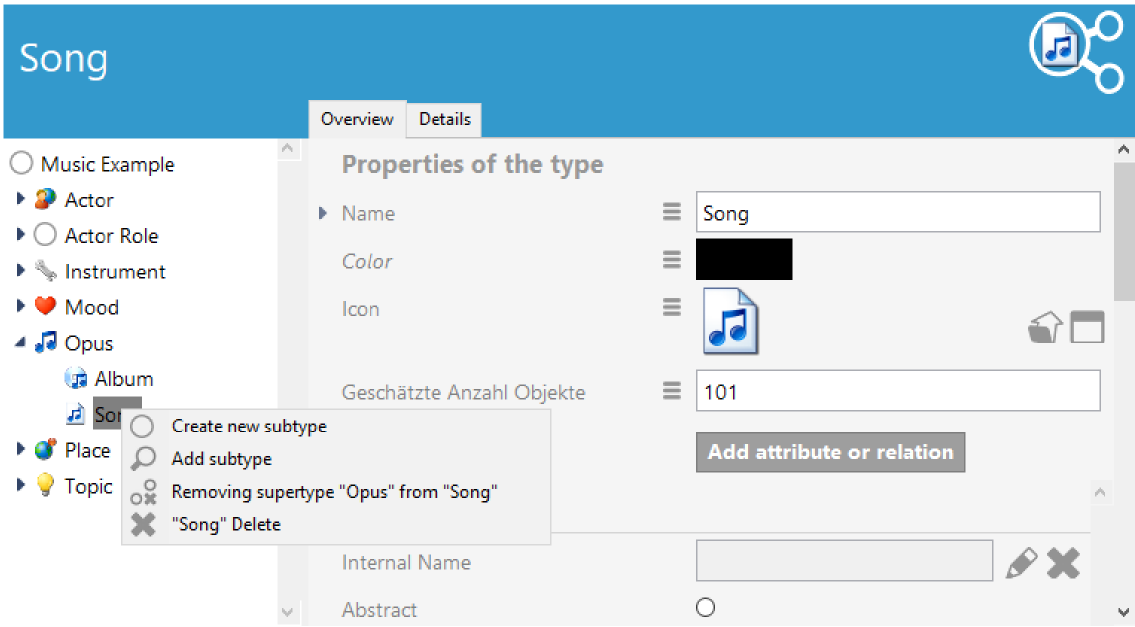
Changing the type hierarchy

In order to change the type hierarchy we have the tree of object types in the main window and the graph editor.

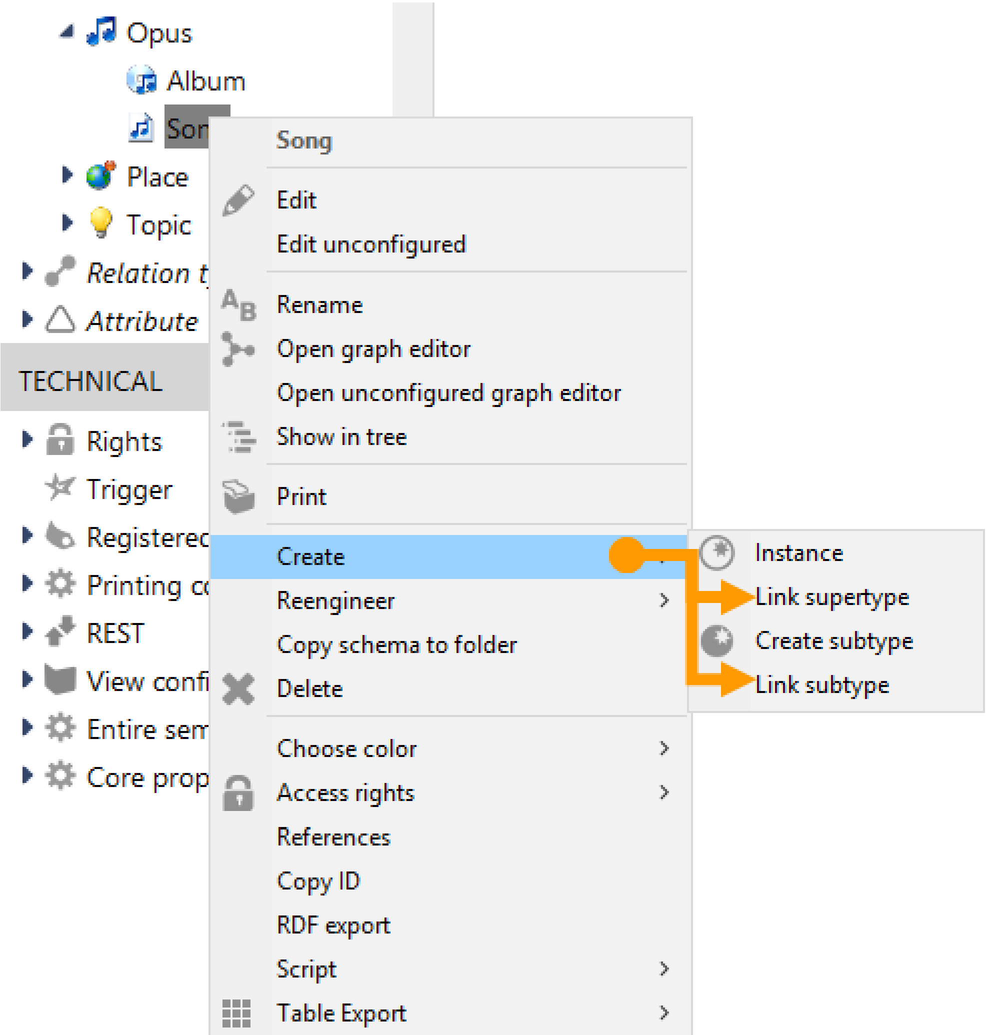
We also can change type assignment when opening the detail editor of the affected type by choosing the options "Edit" or "Edit unconfigured" in the context menu:



In the hierarchy tree of the detail editor, we will find the option "Removing supertype x from y" in the context menu.

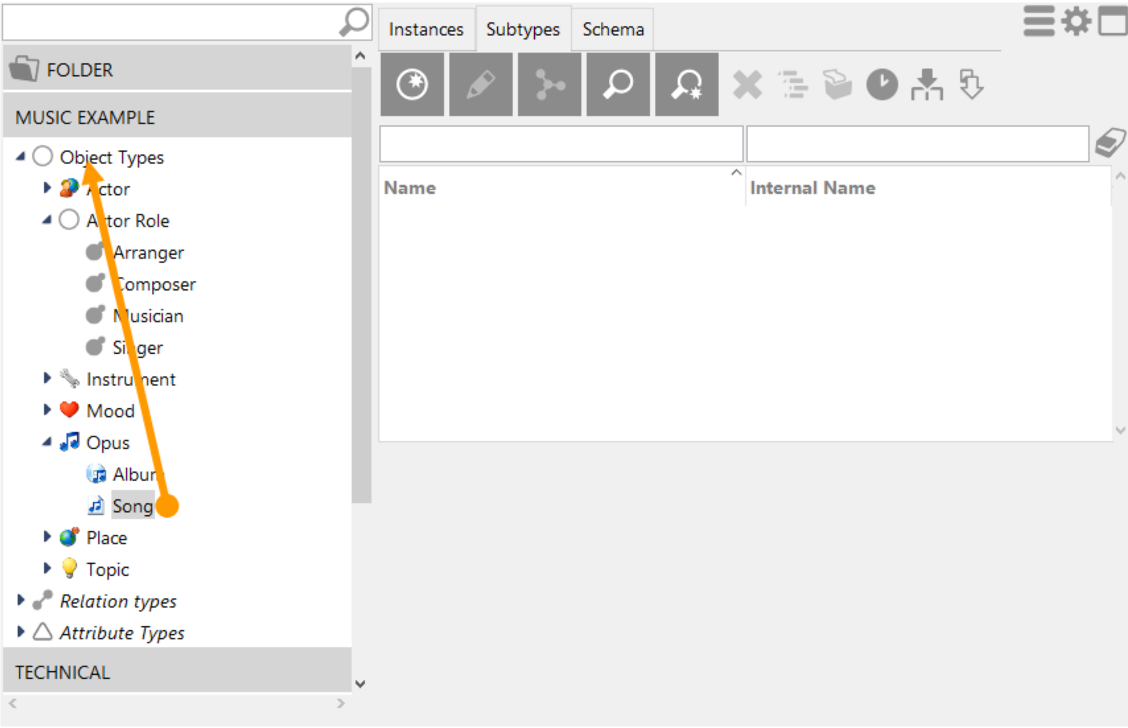


Using this option we can remove the currently selected object type from its position in the hierarchy of the object types. In the organizer, we can link types to other types in order to create multi-hierarchical schema:



Shortcut: By means of drag & drop we can move an object type to another branch of the hierarchy. If we hold down the **Ctrl key** when using the drag & drop function the object type will not be moved but additionally assigned to another object type.

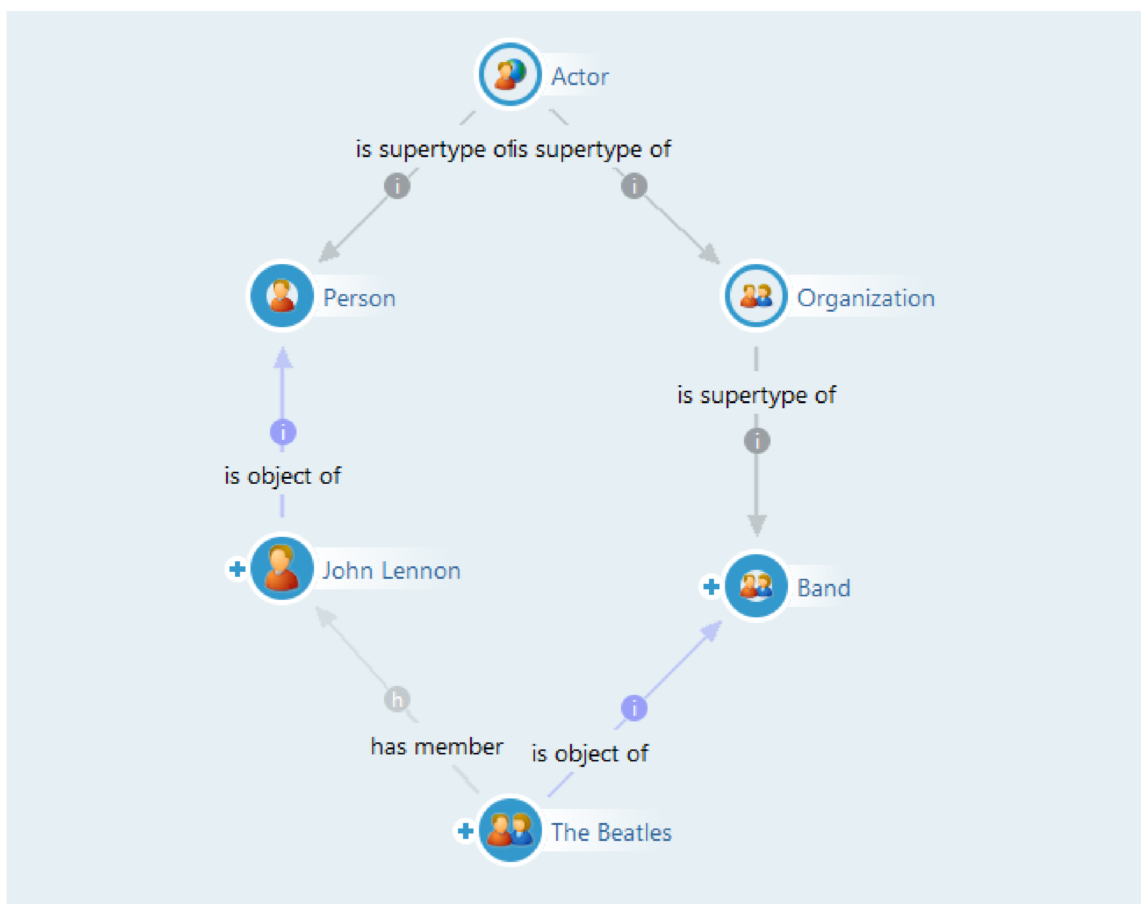
What still applies is: the hierarchy of the object type allows multiple assignments and inheritance.

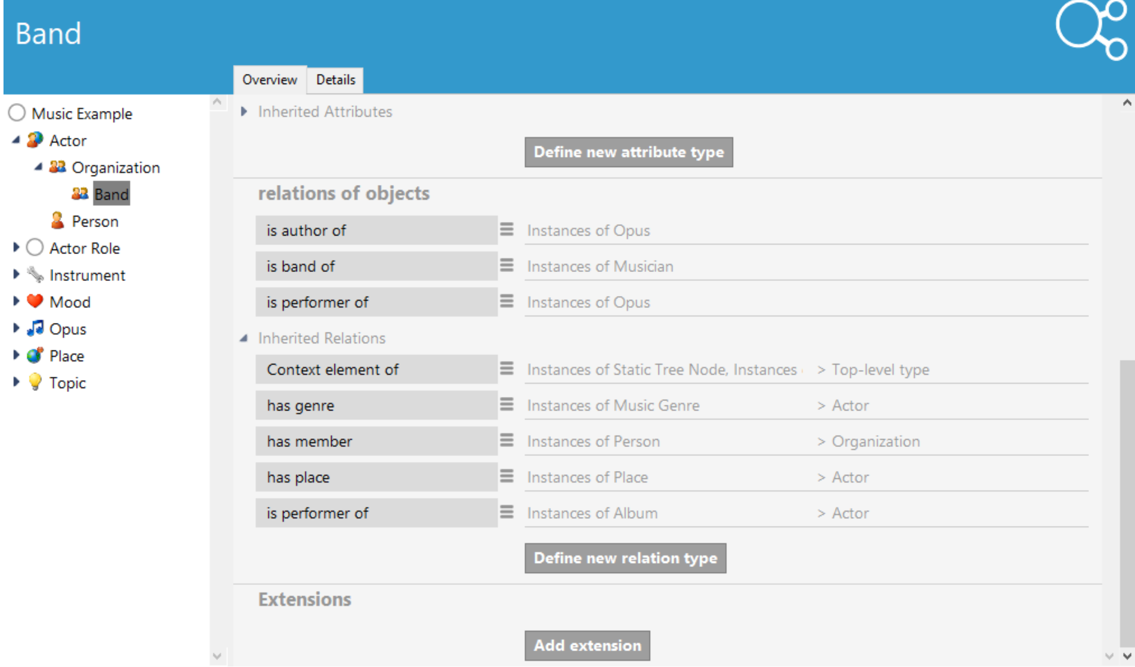


Configuring object types with properties

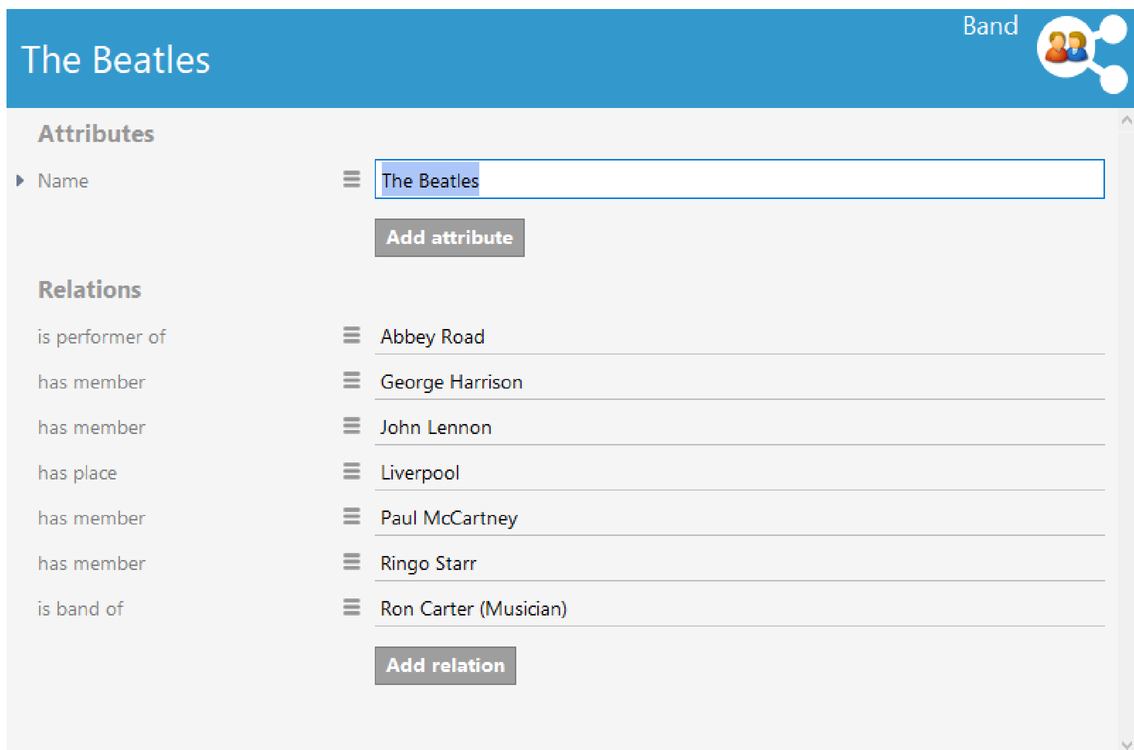
In the simplest case we define relations and attributes with an object type such as "band" or "person" and thus make them available for the specific objects of this type. (For example the year and location the band was established, date of birth and gender of people, location and date of events.)

If the object type for which the properties are defined has more subtypes the principle of inheritance will take effect: properties are now also available for the specific objects of the subtypes. Example: as a subtype of an organisation, a band inherits the possibility of having people as members. As a subtype of "person or band" the band inherits the possibility of taking part in events:





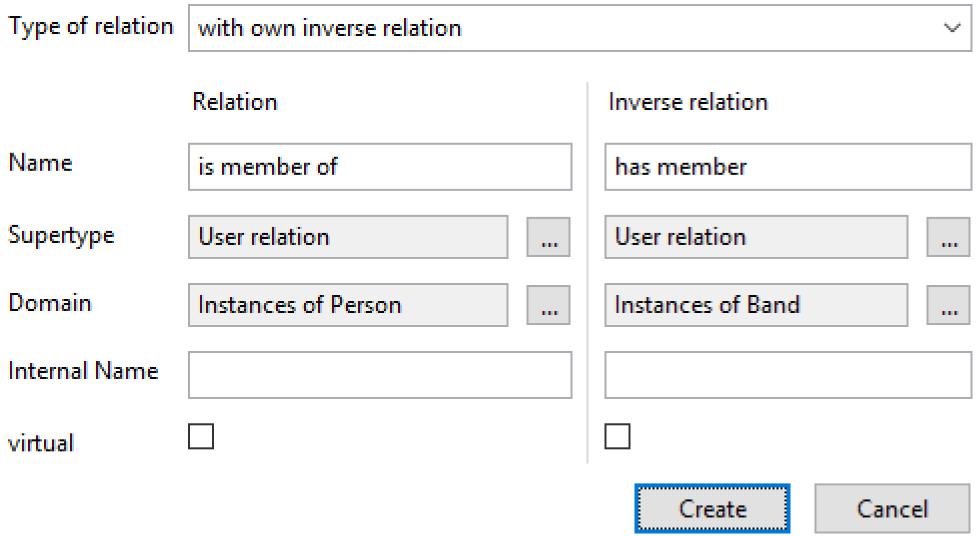
The editor for the object type "band" with directly defined and inherited relations there.



With a specific object the inherited properties are available without further ado and the difference goes without notice.

Defining relations

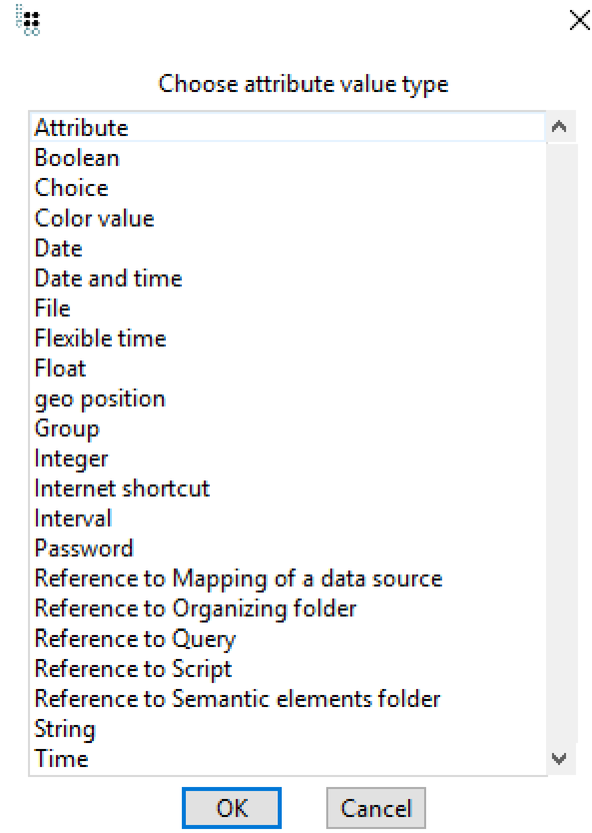
When dealing with relations, the following basic principle governs at i-views: a relation cannot only be unidirectional. If we know of a relation for the specific person "John Lennon" to be "is a member of the band The Beatles" it then implies for the Beatles the contents "it has a member called John Lennon". These two directions cannot be separated. Therefore, i-views demands from us the types of source and target of the relations when creating new relation types — in our example that would be person and band as well as differing names: "is member of" and "has member".



Hence the relation is defined and can now be drawn between objects using drag & drop.

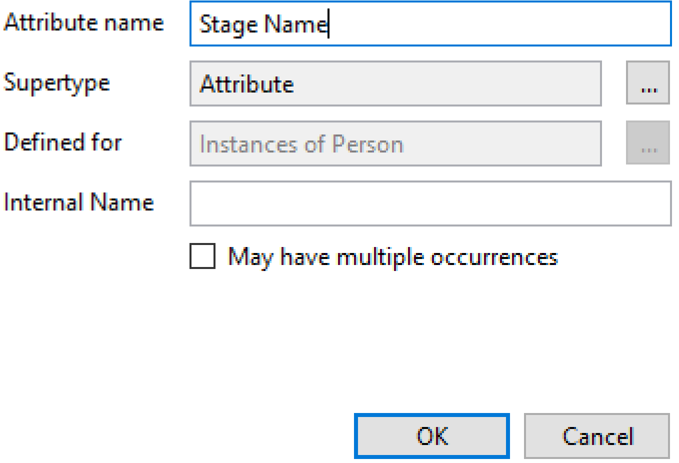
Defining attributes

When defining new attribute types, i-views needs, above all, the technical data type as well as the name.



The intention of using these data types is not to define everything as character strings. Technical data types in a defined format later offer special feasibilities of inquiring and comparing. For example, numerical values may be compared to larger or smaller values within the structured queries and a proximity search can be defined for geographic coordinates, etc.

After having defined the attribute value type, the name of the attribute can be defined:



1.2.2. Relation types and attribute types

Relation types and attribute types (in brief property types) are always properties of specific objects.

1.2.2.1. Creating a new relation type

Via the button "add relation" in the object editor or in the relation type part of the organizer, the editor starts to create a new relation type.

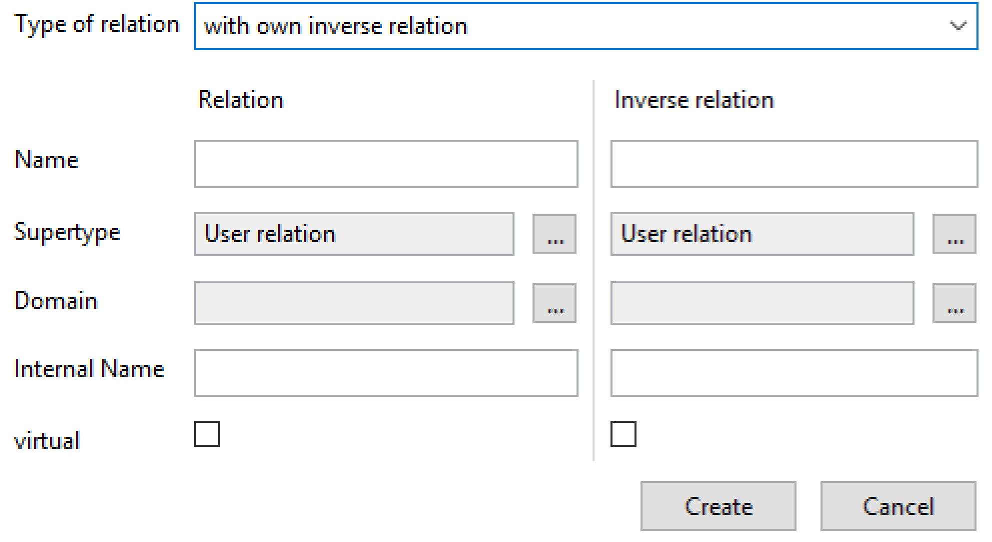


Figure 1. Editor for creating a new relation type

Type of relation

"with own inverse relation" is the default case, for which each relation half as its own name. "Symmetric" is for relations within the same domain only and offers one name for both directions.

Name of new relation

Names for relation types may be chosen freely within i-views but should be selected under the premise of a comprehensible data model. The following convention may be of help for this: the name of the relation is phrased in such a manner that the structure [name of the source object] [relation name] [name of the target object] results in a comprehensible sentence:

```
[John Lennon] [is a member of] [The Beatles]
```

Furthermore it is helpful when the opposite direction (inverse relation) takes on the word selection of the main direction: "has a member / is a member of".

Supertype

Specifies the relation supertype within the relation type hierarchy.

Like object types, relation types and attribute types can be structured within in forms of a hierarchy. The hierarchy of relation types is a simple, but powerful instrument to accommodate the complexity.

Example: For queries, the relation type "has author" can be used to define who has written the song text an who has written the composition. At the same time, we have queries for which we don’t need differentiation and for which all participants need to be requested.

Without relation type hierarchy, all queries would be much more complex because we would have to insert all the relation types fulfilling this circumstance. Instead, we simply can define the relations "writes text" and "writes composition" as subtypes of "writes song" (or: "is author of"). By means of this mechanism, we still can query on the level of "writes song", but due to the inheritance i-views automatically queries the relation subtypes as well.

The subtype therefore implies the supertype. This principle works for relation types and for attribute types or object types.

Domain

Here we define by which object types the relation has to be created: one object type forms the source of the relation and another object type the target. The target object type, in turn, forms the definition area of the inverse relation. To simplify matters, when creating you may only enter one object type at this stage. Afterwards, further object types may be defined in the editor for the relation type (see below).

Internal name

If the relation is intended to be referred by a script, the internal name serves for identification and reference.

Virtual

If we need single-sided relations, we can define which relation half is the single-sided one and which relation half is only virtual. The virtual relation half is only rendered when listing in the relation instances list or can be used for queries. For more information about single-sided relations, see chapter "Single-sided relations".

1.2.2.2. Creating a new attribute type

Via the button "define new attribute" in the object editor the editor starts to create a new attribute type:

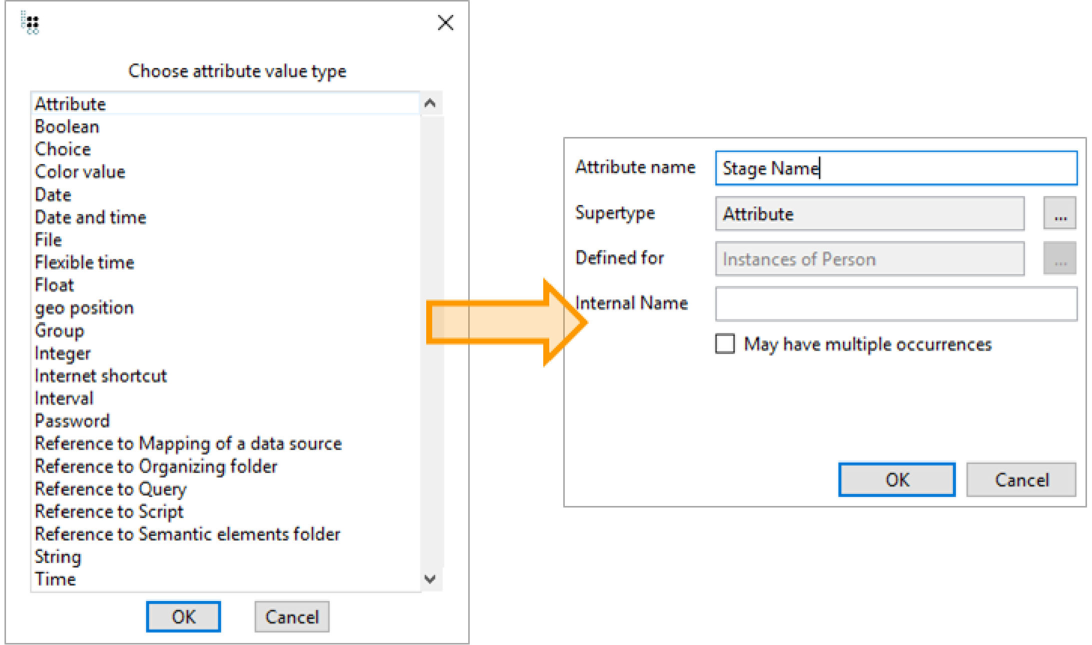


Figure 2. Two-stage dialogue for creating a new attribute type

In the left-hand window the format of the attribute type is defined (date, floating point number, character string, etc.)

The following technical data types are available:

Type of data	What do the values look like?	Example (music graph)
Attribute	abstract attribute, without an attribute rating	
Boolean	"yes" or "no"	music band still active?
Choice	string values which can be selected from a drop-down menu	role; design of a music instrument (hollow body, fretless, etc.)

Type of data	What do the values look like?	Example (music graph)
Colour value	colour selection from a colour palette	
Date	date dd.mm.yyyy (in the German language setting)	publication date of a recording medium
Date and time	date and time dd.mm.yyyy hh:mm:ss	start of an event, e.g. concert
File	random external data file which will be imported into the Knowledge Graph as a "blob"	WAV file of a music title
Flexible time	month, month + day, year, time, time stamp	approximate date when a member joined a band
Float (floating point number)	numerical value with a random number of decimal places	price of an entrance ticket to an event
Geo position (geographical position)	geographical coordinates in WGS84 format	location of an event
Geometry	geometric or geographic shape in (E)WKT format	polygons, lines, points, e.g. shape of a building on a map
Group	without attribute value, serves as a medium for meta attributes to be grouped	
Integer	numerical value without decimal places	runtime of a music title in seconds
Internet shortcut	URL	website of a band
Interval	date interval: interval of numbers, character string, time or date	period of time between the production of an album and its publication
Password	A hashed value (SHA256), which is used to validate the password	
Reference to […]	reference to parts of the Knowledge Graph configuration: search, diagram of a data source, scripts and files	
String (character string)	random sequence of alphanumeric characters	review text to a recording medium
Time	time hh:mm:ss	Starting time of an event

After selecting and confirming the attribute type it can be further specified with the name of the attribute in the subsequent dialogue.

Supertype

Here it is defined at what level in the hierarchy the attribute type should be placed.

Internal name

If the attribute is intended to be referred by a script, the internal name serves for identification and reference.

Defined for

Here it is defined for which types of objects the attributes can be created. In the dialog for creating a new attribute type, only one domain can be defined. Further domains can later be added in the attribute type editor, see Editing details.

May have multiple occurrences

attributes may occur once or more than once, depending on the attribute type: a person only has one date of birth but may, for example, have several academic titles at the same time (e.g. doctor, professor and honorary consul).

1.2.2.2.1. Attribute types in Detail

This section describes implementation limitations of individual attribute types as well as those that are specified in more detail via additional configuration options.

Choice

Choice attributes have exactly one selection from a list of predefined options as value.

Entries

The possible choice options are defined in this list. Translations and order of the options can be specified later with the schema builder, see Editing details.

1.2.2.2.2. Date and Time, Time

Please note that storage is accurate to the second.

File

Storage

Files can be saved either directly in the database or in an external blob service.

Flexible Time

Flexible time is useful wherever exact values are unknown or irrelevant. Historical dates such as “200 BC” are also possible. The time formats specified in the schema can be used to define which time entries are accepted when the user enters data. Please note that storage is accurate to the

minute at best. Formats that do not include a year are recurring times. Searches or comparisons with such formats are semantically questionable, which is why their use is not recommended. Future versions of i-views will no longer support recurring times.

Geometry

Geometry attributes store geometric shapes (e.g. points or lines in a Cartesian coordinate system) and geographic shapes (e.g. areas on a map) in the *Well Known Text (WKT)* format, which is also supported by common geographic information systems (GIS) and other database systems (e.g. Elasticsearch).

Point
POINT (10 10)

Line string
LINESTRING (10 10, 20 20, 30 40)

Area
POLYGON ((10 10, 10 20, 20 20, 20 15, 10 10))

Area with hole
POLYGON ((0 0, 0 20, 20 20, 20 0, 0 0),(5 5, 5 15, 15 15, 15 5, 5 5))

For geographical forms, the geographical reference system is also defined in the form of an EPSG identifier. The most common reference system WGS84 worldwide corresponds to the EPSG identifier 4326, for example. Depending on the reference system, the position data have to be defined in degrees or meters.

NOTE In WKT, positions in degrees are always specified in the order longitude, latitude, while in everyday life the order latitude, longitude is common.

Geographical position 49.87283° N, 8.6512° E
SRID=4326; POINT (8.6512, 49.87283)

Rectangle around Munich in the UTM system
SRID=25832; POLYGON ((679578 5325187, 702805 5325961, 702165 5344040, 679011 5343266, 679578 5325187))

Geometry attributes also accept GeoJSON as input, as well as typical representations of geographic points like latitude/longitude, MGRS and UTM. The spacial reference is automatically derived in these cases.

Format	Input	Output
Lat/long	N49° 52' 18" E8° 39' 0.96"	SRID=4326;POINT(8.6502666667, 49.8716666667)
UTM	4Q FJ 1 6	SRID=32604;POINT(610000, 2360000)
GeoJSON	`{ "type": "LineString", "coordinates": [[8.62, 49.86], [8.64, 49.87]] }`	SRID=4326;LINESTRING (8.62 49.86, 8.64 49.87)

When a new geometry attribute type is created, the following additional configuration options are available:

Is measured

Defines whether the shapes have an additional third coordinate component M which represents any measured value.

```
POINT M (10 10 42)
```

Has z coordinates

Defines whether the shapes have an additional third coordinate component that represents the height. For geographic data, this is the height above normal zero in meters.

```
POINT Z (10 10 300)
```

Z and M can also be combined. Points then consist of four coordinate components, the third of which expresses the z value and the fourth m value.

```
POINT ZM (10 10 300 42)
```

Has geographic reference system

Defines whether the data are geographic data.

Limit to reference system

A comma-separated list of EPSG identifiers can be specified here. Only values to which one of the allowed identifiers has been assigned are accepted.

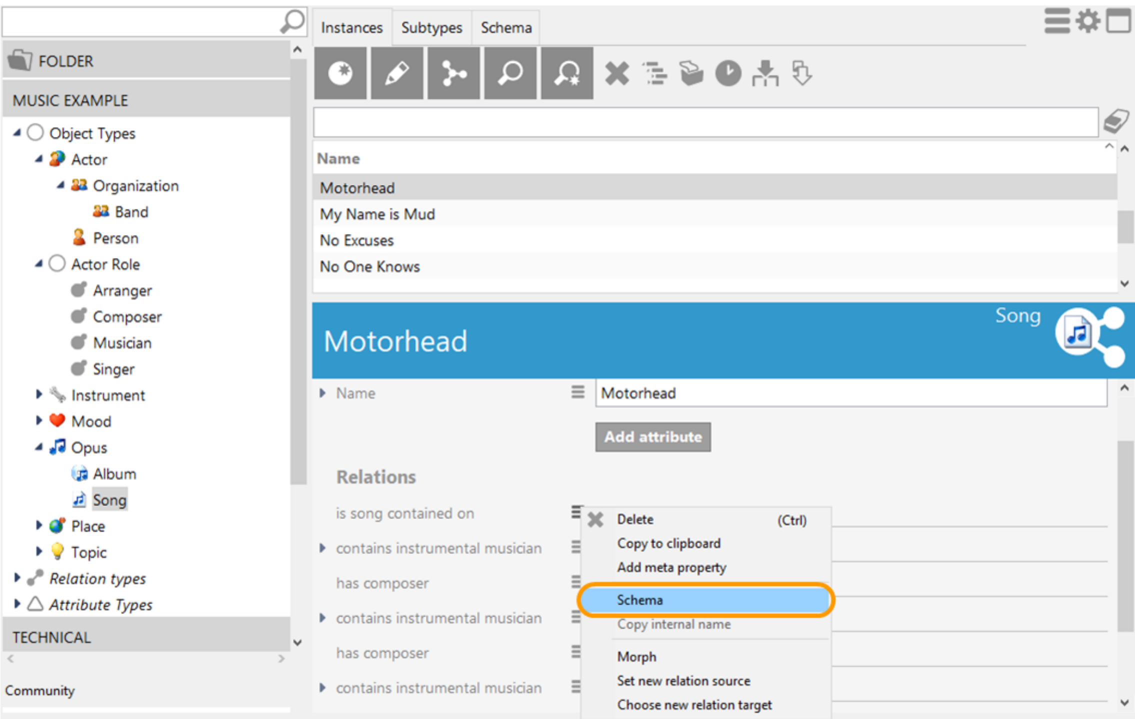
Limit to shapes

Here you define which shapes can be saved in the attribute. If no shape is specified, all forms are allowed.

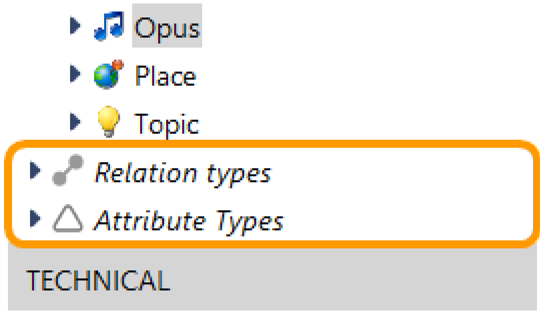
1.2.2.3. Editing details

The dialogs for creating new attribute and relation types are limited views of the attribute and relation type editors. To edit details of relations and attributes, editors must receive and enhanced scope of functions.

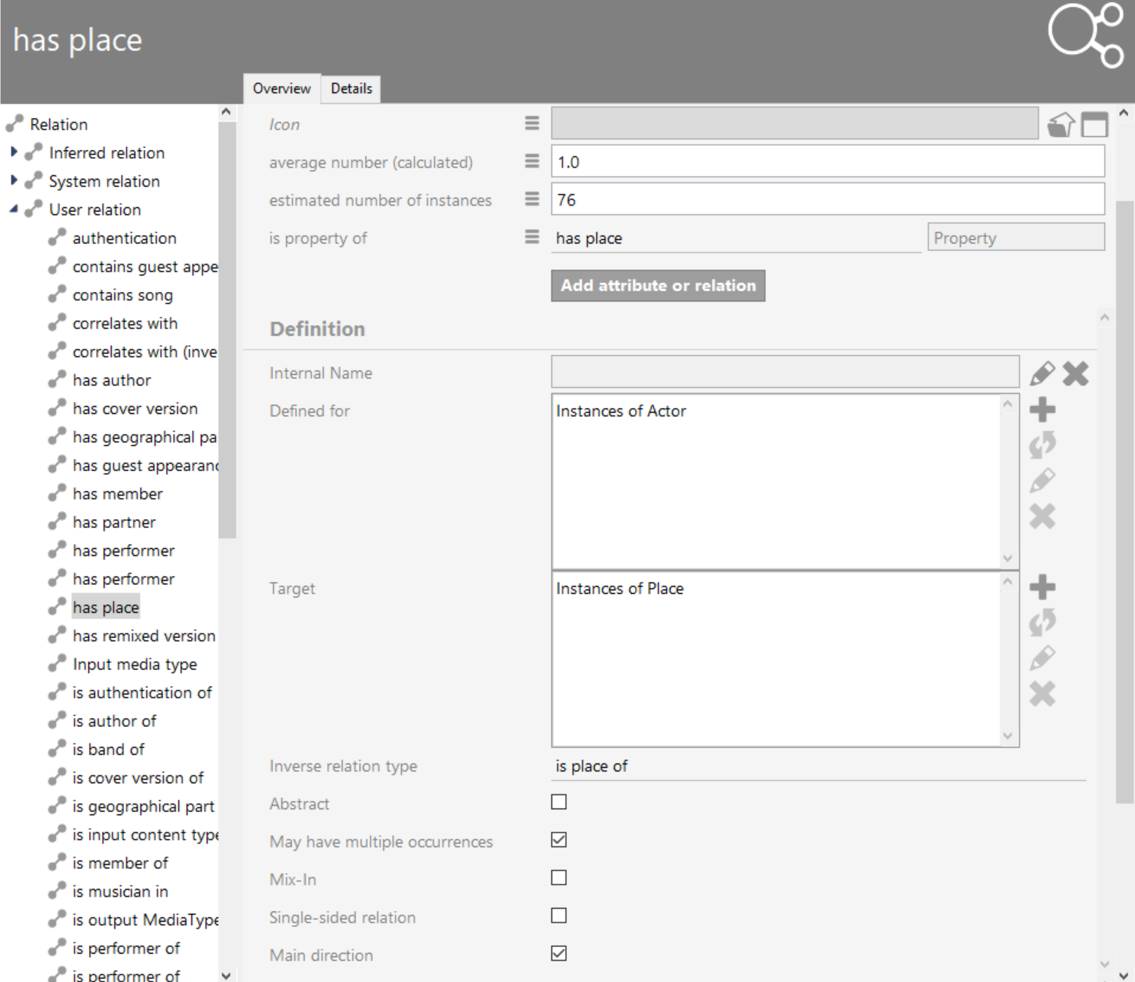
You get to these two editors via the listing of relations and attributes on the "Schema" tab of the object editor:



Alternatively, you can use the hierarchy tree on the left side of the main window for access. The hierarchies for relation and attribute types are located underneath the object types. The editors are started by right-clicking on the relation or attribute to be edited in the context menu and choosing "Edit" .



Next, we will look at the details of the definition of properties by using the relation type editor as the example (the attribute type definition is a subset thereof):



Defined for

Here we can subsequently check for which object types the relation can be created. Relations can be defined between several objects and thus have several sources and targets. In this way, we can allow persons and bands to be authors of a song in the schema or assigned a location — even if they do not have a super-type in common.

We can use the "Add" button to add additional objects. We can use "Remove" to prevent this object type and all its objects from entering into this relation. "Change" makes it possible to replace an object type. Already existing relations are then deleted by the system. If there are relations to be deleted, a confirmation prompt appears before the change is made.

Target

Here you can change retrospectively for which types of objects the relation can be used. To change the target object type you have to switch to the inverse relation type: The button for changing bears the label of the inverse relation type. After clicking on the button, the inverse relation appears in the editor and can be edited in the same way as the previous relation.

Abstract

If we want to define a relation which is only used for grouping but is not supposed to define concrete properties, we define it as "abstract".

Example: If the relation "Writes song" is defined as abstract, this means: if we create songs and their relation to artists and bands, we can now enter specific information (who wrote the lyrics, who wrote the music). The unspecified relation "Writes song" cannot be created in the actual data but can only be used for queries.

May have multiple occurrences

One characteristic of relations is whether they may have several occurrences. For example: the relation "Has place of birth" can only occur once for each person whereas e.g. the relation "is member of" can occur several times for a person. Hence, logical matters can be modeled precisely. For example, musicians as persons can only have one place of birth but (at the same time) can also be members of several bands. Whether the relation can occur multiple times is specified independently for each direction of the relation: A person can only have one place of birth but the place can be the place of birth of several persons.

The option can only be deactivated if the relation does not occur several times in the actual data set. If it occurs several times, the system cannot decide automatically which of the relations is to be removed.

Mix-in

Mix-ins are described in the Extension chapter.

Single-sided relation

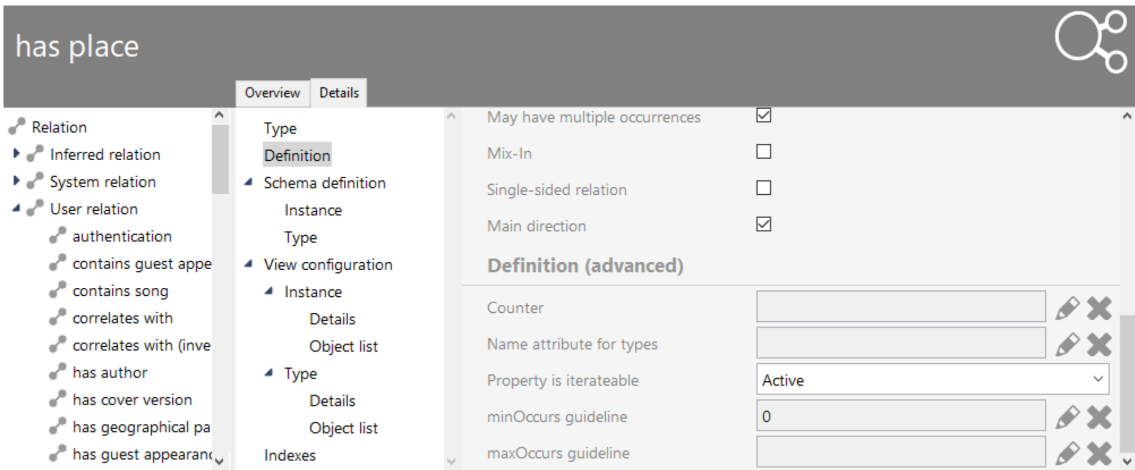
If the relation direction combines many source objects into one "hot spot" object, a single-sided relation is used to improve access performance. Further information on this can be found in the sub-chapter on "Creating a new relation type".

Main direction

Every relation has an opposite direction. In the core, the two directions are equivalent, but there are two places where it makes sense to determine a main direction:

- In the Graph editor: Here the relations always present themselves in the main direction in relation to the direction of the arrow and labeling; irrespective of the direction in which they were created.
- For single-sided relations (without inverse relation)

Additional setting options for relations and attributes are located in the "Definition" sub-item on the "Details" tab. The setting options under Definition are often used and that is why they are already available on the Overview tab. Under "Definition (advanced)" in contrast, there are setting options that are not required as frequently.



Counter

If a number is entered in the counter, this is the number with which objects of this type are counted up. The JavaScript functions getCounter(), increaseCounter() and setCounter() can be used to access the counter.

Name attribute for objects

Typically many views in i-views only represent an object via its name (e.g. in object lists, hierarchies, in the Graph editor, the relation target search, etc.). Instead of the name you can use any other attribute of the objects here with which it can be represented. A prominent example for products: The article number.

NOTE	Can only be set on object types, not relation or attribute types

Name attribute for types

This can be used also to select an alternative attribute for a more descriptive display for types.

Property is iterable

Selection options: Active / Write only / Inactive. Default: Active.

Sometimes the maintenance of the index for iterating properties severely affects performance. This typically happens with meta properties such as "changed by" or "changed on" which do not necessarily have to be taken into account all the time. In such cases we recommend setting the properties to cannot be iterated by using the "Inactive" selection option. The purpose of "Write only" is to deny read access but still allow write access. This makes it possible to test for inadvertent side effects.

minOccurs guideline

Specifies the minimum number of times a property is supposed to occur on an object. If the number falls below the specified number, the property is displayed in red in the user interface but the object can continue to exist. An import ignores the reference value.

maxOccurs guideline

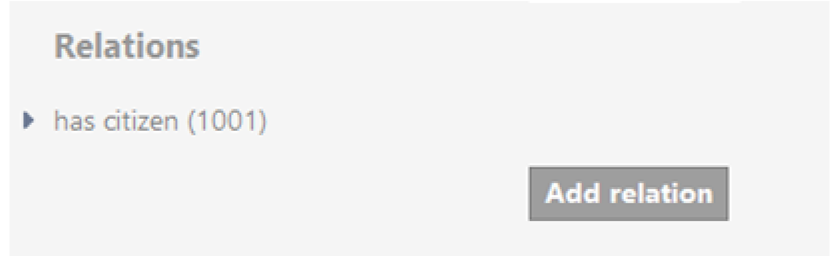
Specifies the maximum number of times the property should occur on an object. If the specified

number is reached, no additional properties can be created. An import ignores the reference value.

1.2.2.4. Single-sided relations

1.2.2.4.1. Application of single-sided relations — basic principles

When an object is called up for import purposes or displaying in view configuration, all of its properties will be loaded (especially when not indexed sufficiently). This in turn means that besides of the attribute values, all existing relations will be loaded including their target objects as well, leading to an overhead which slows down performance.



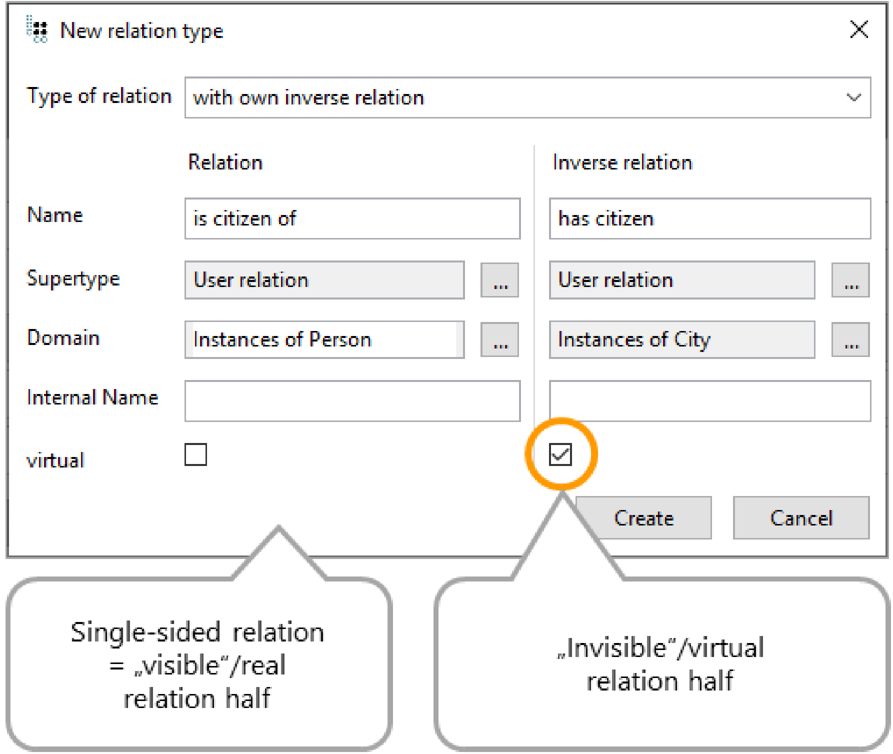
Especially for *catalog objects*, the loading all properties can lead to long loading duration. A catalog object is an object which serves as central reference for other objects and therefore is interrelated with them.

A Knowledge Graph has objects of the type "city" which are connected by relations to its citizens. When a detailed view of a city has to be loaded for indicating the number of citizens only (and not their names, addresses and hobbies etc.), single sided relations make sense for this purpose.

In this case, the single-sided relations direct from the individual satellite objects towards the catalogue object. This results into the relation "is citizen of" being visible on the citizen side only, but the relation "has citizen" from the city towards the citizens will be suppressed. Nevertheless, the "virtual" relation "has citizen" can be used for structured queries and it can be found within the schema.

1.2.2.4.2. Defining single-sided relations

In order to define a single-sided relation, we must specify in the dialog which relation half (original or inverse orientation) has to be kept virtual, in other words "invisible". Here fore we choose the checkbox "virtual" on the affected half. The other relation half automatically becomes the real relation half which builds up the relationship between start domain and target domain.

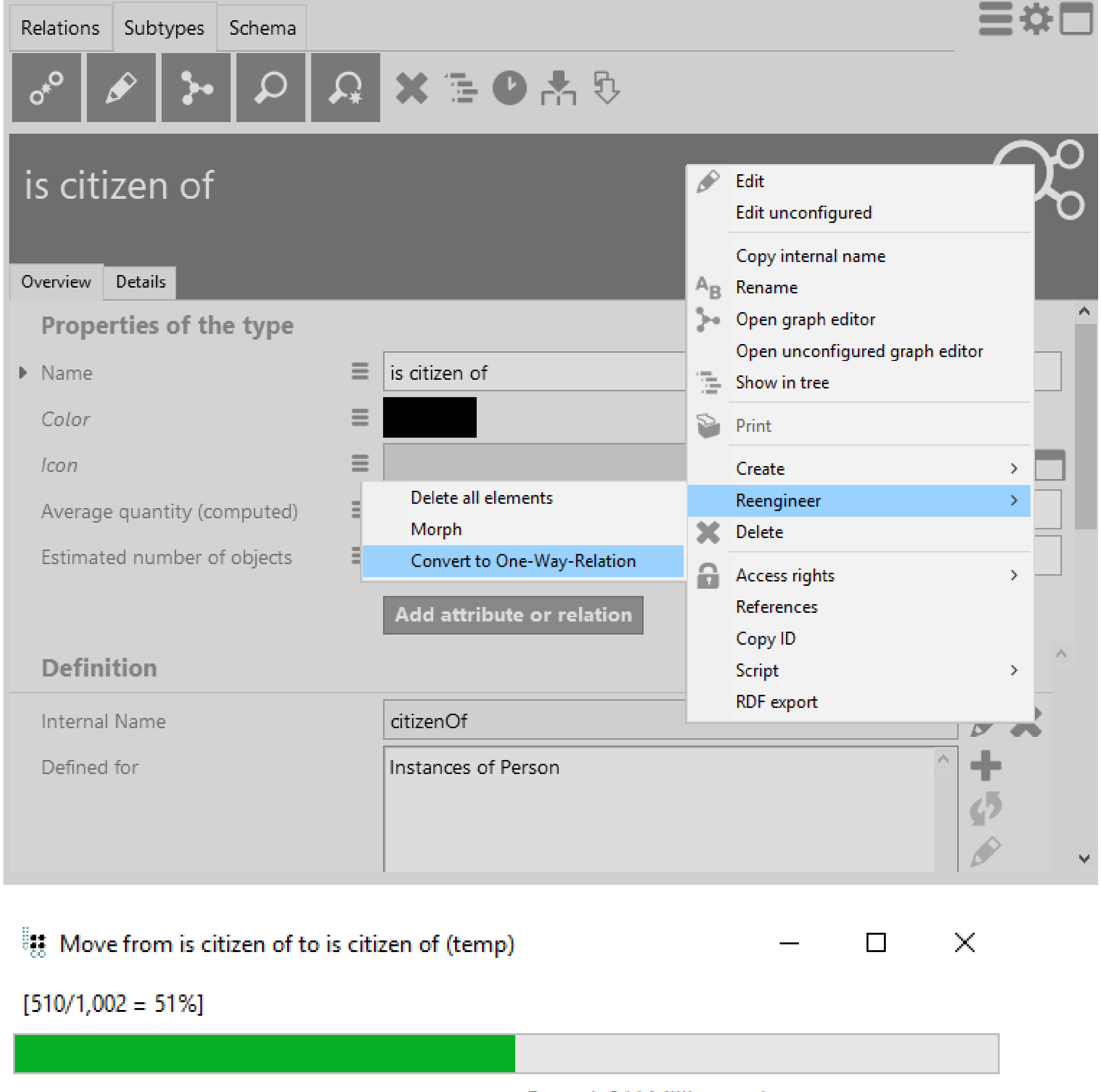


1.2.2.4.3. Supplementary declaration of a conventional relation as a single-sided relation

When a preliminary declared conventional relation type is going to be converted into a single-sided relation type, the instances of the virtual relation half will be deleted. This process can be inverted when redefining the relation form. Then the particular relation halves are going to be determined again.

The conversion to single-sided relations will show its effect as follows: For a catalog object, all the virtual relation halves including their relation targets are not going to be displayed anymore, but the virtual relation instances are still rendered as an instance in the Knowledge Graph and therefore can be called up in structured queries.

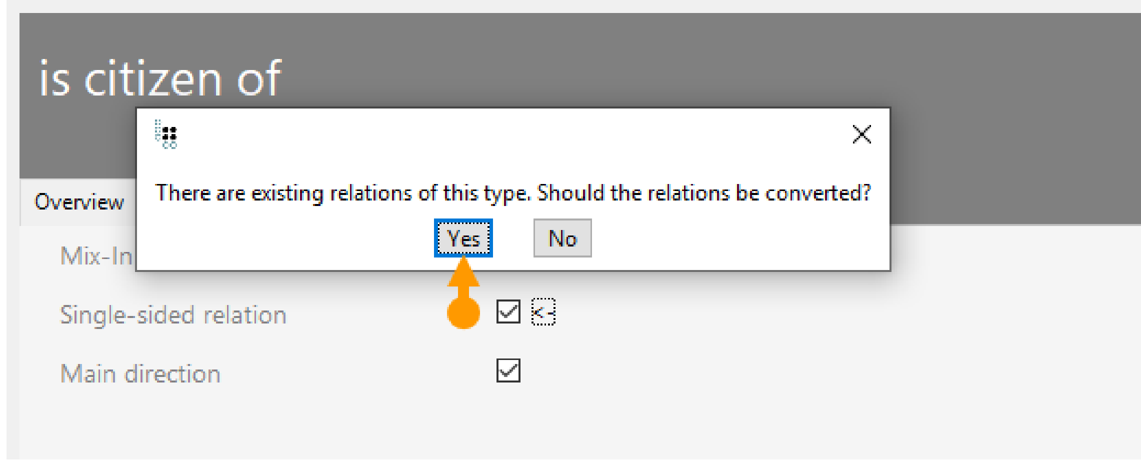
In the best case, when defining import mappings for large amounts of objects that relate to a catalog objects, always use the real, single-sided relation type half. This can lead to performance improvements when importing.



As a result, the checkbox "Single-sided relation" indicates that the respective relation half is used as a single-sided relation.

NOTE

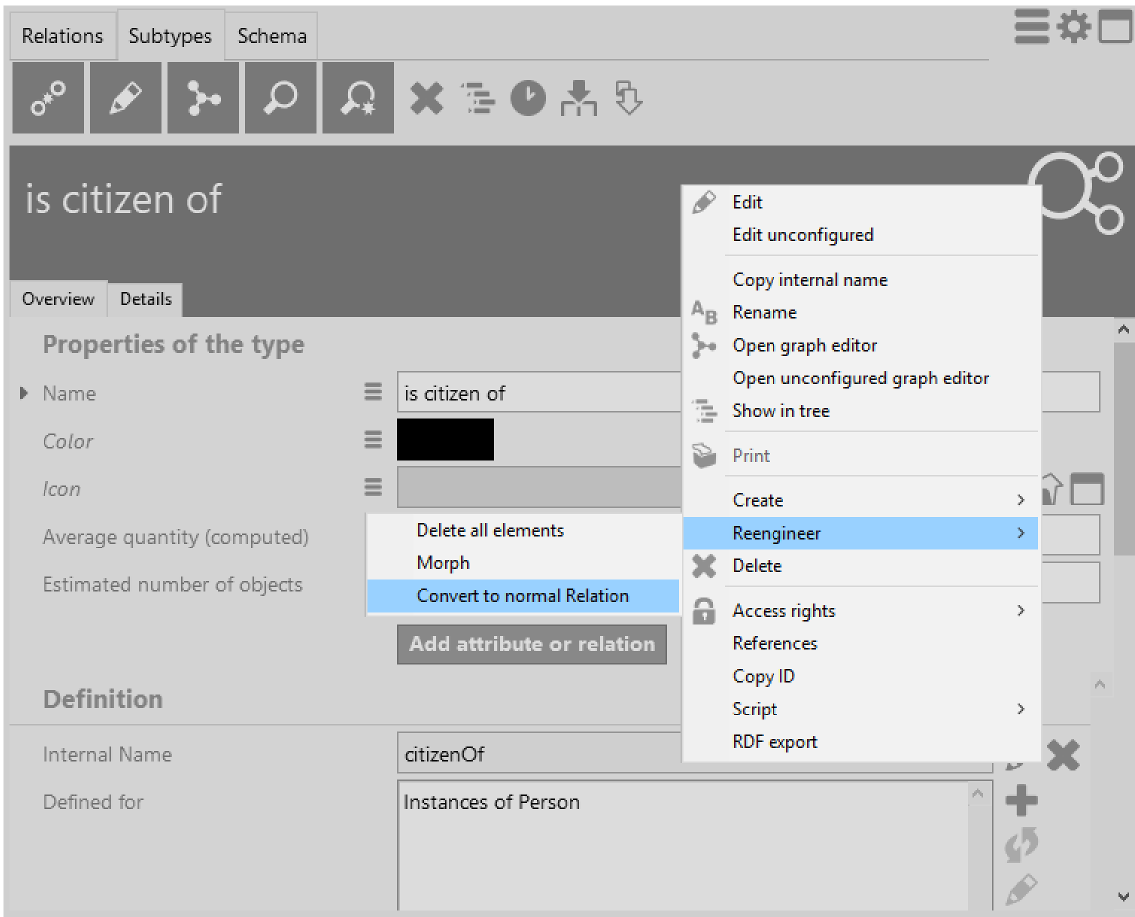
Until i-views 5.3, the checkbox of the Boolean attribute "Single-sided relation" only served for indication purpose. Since i-views 5.4, a redefinition only can be executed by clicking on the checkbox **or** via the context menu in the detail editor.



NOTE After conversion to single-sided relation, the performance for indicating **virtual** relations can be improved by means of indexing.

1.2.2.4.4. Supplementary conversion of a single-sided relation into a conventional relation

If we realize afterwards that a relation type actually should be declared as a conventional relation type, a correction can be made without further consequences. In the detail editor of the relation type, we therefore click onto the context menu and choose Reengineer > Convert to normal relation or we deselect the checkbox "Single-sided relation".



Immediately, the Knowledge-Builder changes all existing virtual and single-sided relations into normal relations.

1.2.2.4.5. Supplementary swapping of the orientation of a single-sided relation type

The supplementary change of orientation of the single-sided relation type is done analogous via the "Reengineer" command in the context menu of the detail editor or by swapping the checkbox selection. In order to do this, we change to the opposite relation type half which has to be converted from virtual to single-sided and choose Reengineer > Convert to one-way relation or we tick the checkbox "Single-sided relation".

1.2.3. Model changes

In i-views you can make changes to the runtime of the model:

- implement new types
- make random changes to the type hierarchy (without creating tables and giving any thought to primary and secondary keys).

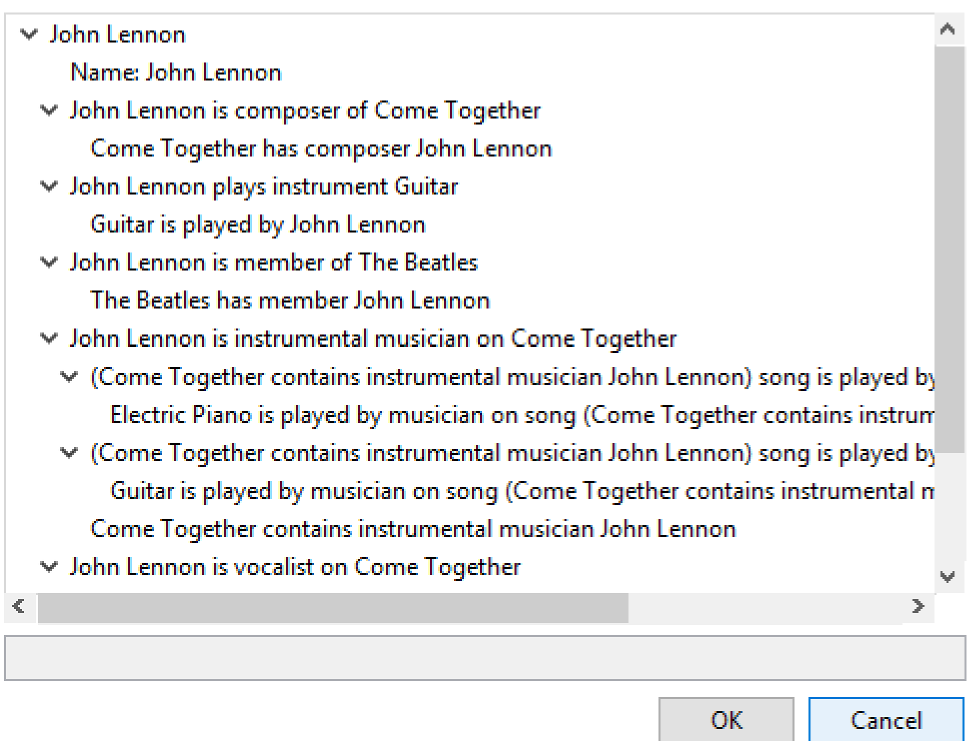
The system ensures consistency. When creating objects and properties the opposite direction of a relation is always included. Attribute values are checked as to whether they match the defined technical data type (for example, in a date field we cannot enter any random character string).

Consistency is also important when deleting: dependent elements always have to be deleted with them so that no remaining data of deleted elements stays in the Knowledge Graph.

- Thus, when an object is deleted all its properties will be deleted along with it. If, for example, we delete the object "John Lennon" we also delete his date of birth and his biography text which we can have as a free text attribute for each person, etc. Likewise, his relation "is member of" to the Beatles and "is together with" to Yoko Ono. The objects "The Beatles" and "Yoko Ono" will not be deleted; they only lose their link to John Lennon.
- When deleting a relation the opposite direction is automatically deleted with it.

Since i-views always ensures that the objects and properties are in accordance with the model, deleting an object type or, where necessary, an operation has far-reaching consequences: when an object type is deleted, all its specific objects are also deleted — analogue to the relation and attribute types.

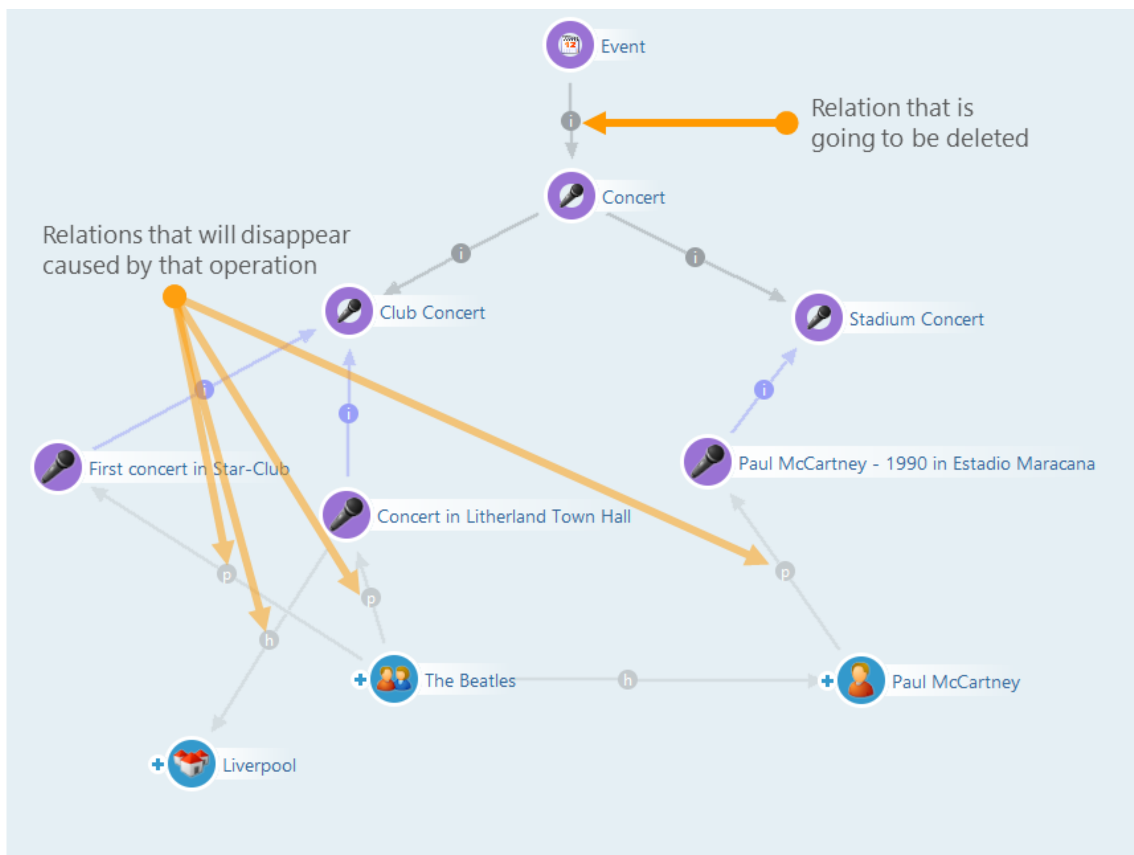
In this process, i-views always provides information on the consequences of an operation. If an object has to be deleted, i-views lists all properties which will thus be removed in the confirmation dialogue of the delete operation:



i-views controls where, by the change, objects, relations or attributes become lost. The user is made aware of the consequences of the deletion.

Not only the deletion, but also conversion or change of the hierarchy type may have its consequences. For example, when objects have properties which no longer comply with the model

after a change in type or change in the inheritance.



Let us assume that we delete the relation "is supertype of" between "event" and "concert" and thus remove the object type "concert" and all its subtypes from the inheritance hierarchy of event to add them to "work", for example. In this case, i-views draws our attention to the fact that the "has participants" relations of the specific concerts would be omitted. This relation is defined in "event" and would thus no longer apply to the concerts.

There are possibilities for preventing the omission of relations as a result of model changes. If an object type has to move within the type hierarchy, for example, the model of the affected relation has to be adapted prior to this.

For example, if "concert" is to be located under "work" within the hierarchy and no longer under "event". To this end, the relation "has participants" will be assigned to a second source: that can be either the object type concert itself or the new item "work". The relation will hence not be lost.

i-views pays particular attention to the type hierarchy. If we delete a type from the middle of the hierarchy or remove a super/sub relation type, i-views then closes the gap which has ensued and puts back the types which have lost their supertypes into the type hierarchy to the extent that they keep its properties as far as possible.

If changes are made to the model, consideration should always be given to the fact that restoring a previous condition may only be carried out by installing a backup. Analogue to the related databases there is no "reverse" function.

1.2.3.1. Special functions

1.2.3.1.1. Change type

Objects already in the Knowledge Graph may be moved to objects of another type. For example, if the object type "event" differentiates to "sports event" and "concert". If there are already objects of the type sports event or concert in the Knowledge Graph, they may be selected from the list in the main window and quite simply moved to a new, more suitable object type using drag & drop.

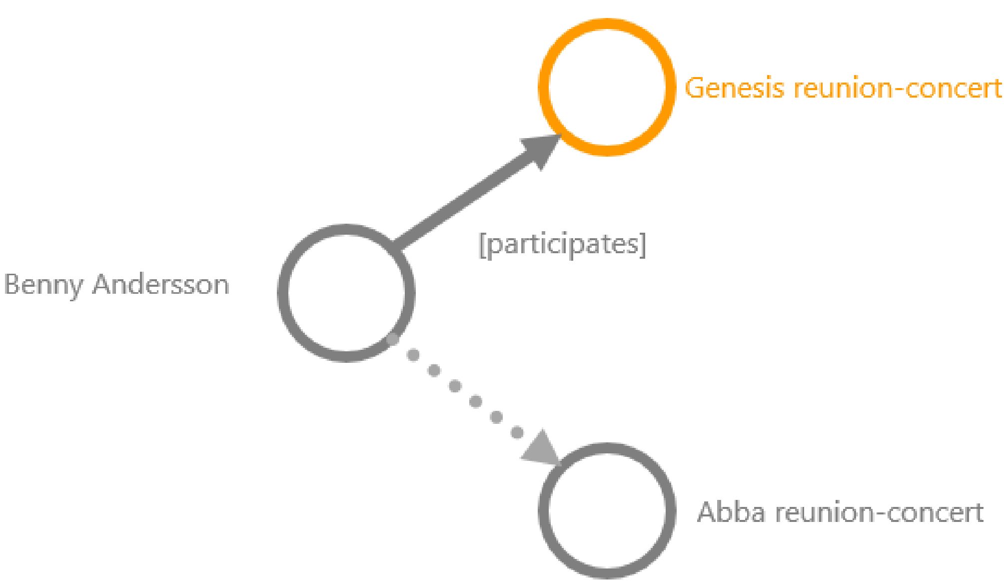
Alternatively, we can find more information in the context menu under the item "edit".

1.2.3.1.2. Select type

Using this operation we can assign a property to an object.

1.2.3.1.3. Choose new relation target

In relations this does not only apply to the source, but also the relation target.

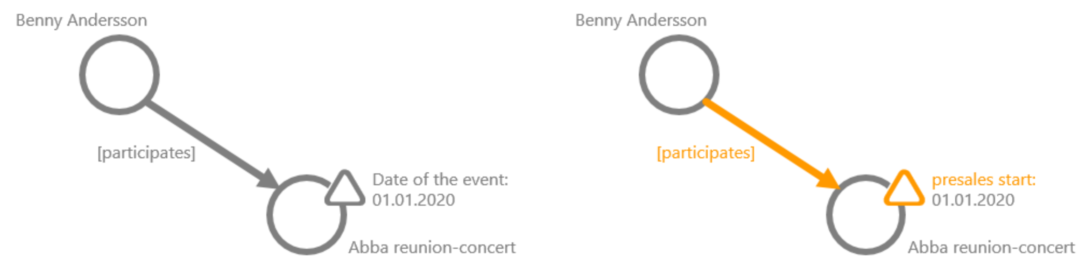


1.2.3.1.4. Convert subtypes to specific objects (and vice versa)

The border between object types and specific objects is, in many cases, obvious but not always. Instead of setting up only one object type called "musical direction" as in the case of our sample project, we could have set up an entire type hierarchy of musical directions (we decided against this in this Knowledge Graph because the musical directions classify so many different things such as bands, albums and songs and therefore they do not provide any good types). It may happen, however, that we change our minds in the middle of the modelling. For this reason, there is the possibility of changing subtypes into specific objects and specific objects into subtypes. Any relations which may already exist will be lost in the process if they do not match the new model.

1.2.3.1.5. Morph

This function converts properties of one property type to another. For relations, source and target of the relation will remain the same, only the relation type will be converted. For attributes, the source will remain the same but in some cases the value type can be changed (e.g. when converting an integer attribute to a float attribute).



When converting the individual relations we are usually quicker when we delete these and replace them with another one. However, it may happen that meta properties are attached to the properties which we do not want to lose. On the other hand, the converting operations are also available for all properties of a type or a selection thereof. A prerequisite is, of course, that the new relation or attribute type is also defined for the source and target objects.

1.2.3.1.6. Schema reengineering for geometry attributes

For geometry attribute types, the following additional functions are available in the *Reengineer* submenu:

Add geographic reference

Converts a geometric attribute to a geographic attribute. The identifier of a spatial reference system has to be specified. The values are adopted without conversion, i.e., `SRID=4326;POINT(9 50)` (50° N 9° E) is adopted from an attribute with the value `POINT(9 50)`.

Remove geographic reference

Converts a geographic attribute to a geometric attribute. The values are adopted without conversion, regardless of their spatial reference. For example, an attribute with the value `SRID=4326;POINT(9 50)` becomes `POINT(9 50)`, and a value of `SRID=3857;POINT (1001875 6446275)` becomes `POINT (1001875 6446275)`.

Add/remove measurement

Adds an M coordinate component or removes existing measurement values. Removed values cannot be restored! When adding, you can specify a default value that all existing attributes receive.

Add/remove z coordinate

Adds a Z coordinate component or removes existing Z coordinates. Removed values cannot be restored! When adding, you can specify a default value that all existing attributes receive.

Convert geographic reference

This allows existing geographical attributes to be converted into a new spatial reference system.

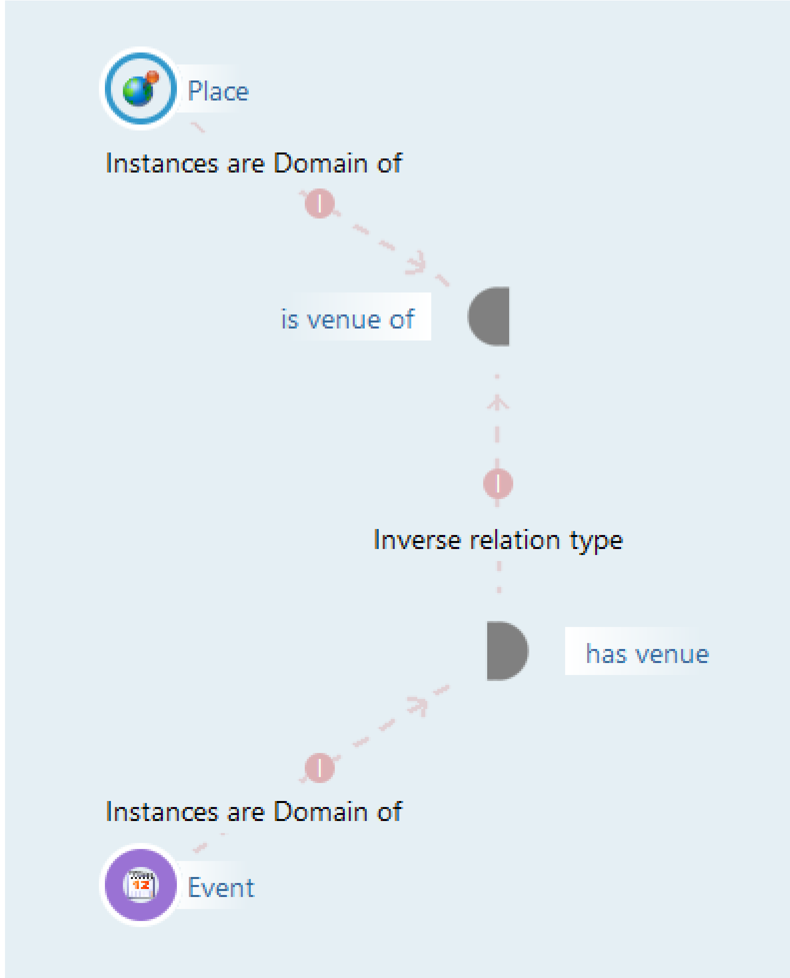
NOTE The external software library *proj* is required for this function. Further information can be found in the technical handbook.

1.2.4. Representation of schema in the graph editor

Until now we have mainly been dealing with linking of specific objects within the graph editor. Presenting such specific examples, discussing them with others and, where necessary, editing them is also the main function of the graph editor. We can, however, also present the model of the Knowledge Graph directly using the graph editor, e.g. the type of hierarchy of a Knowledge Graph.

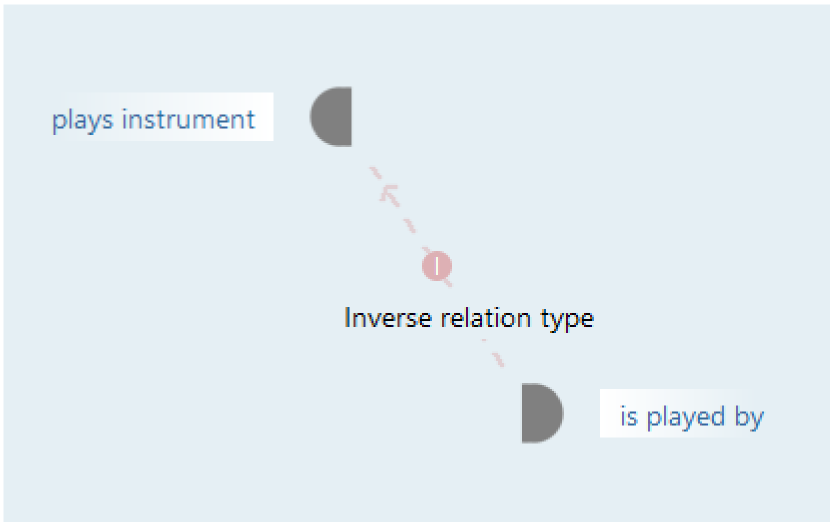
Types of objects will then be displayed as nodes with a coloured background and types of relations

as a dotted line:

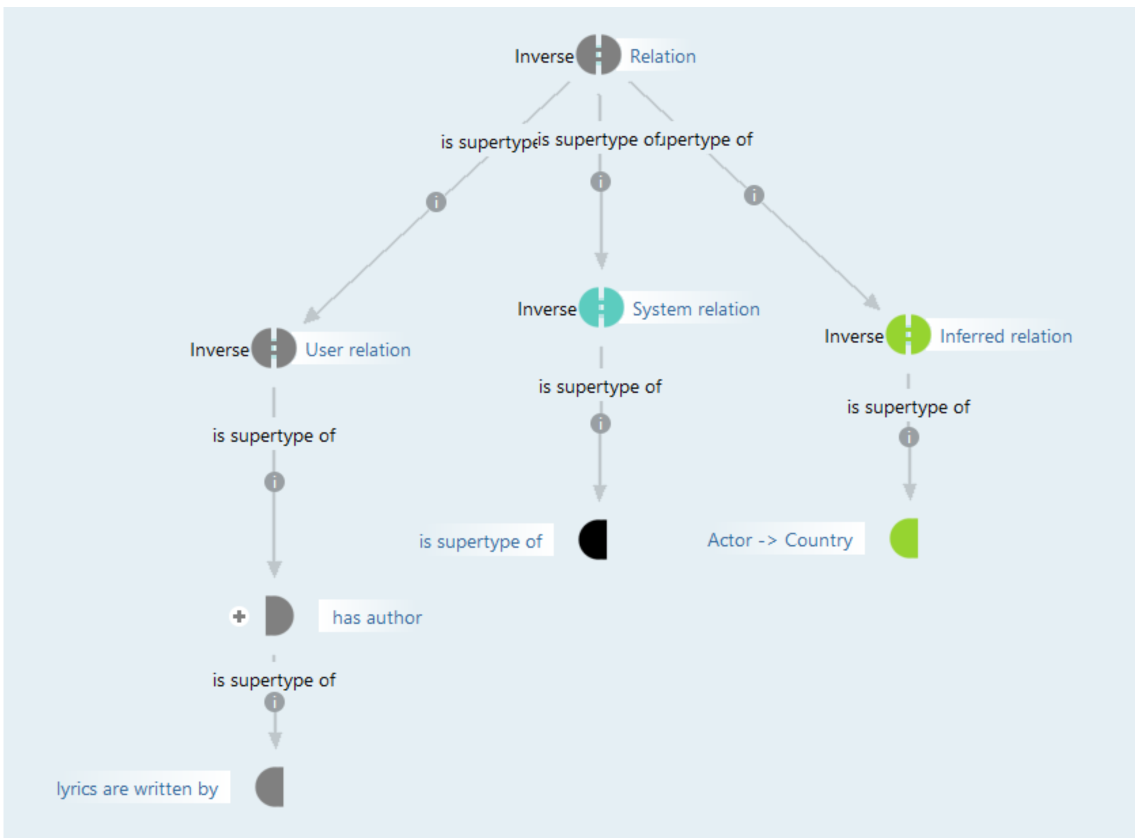


Relation types in the graph editor

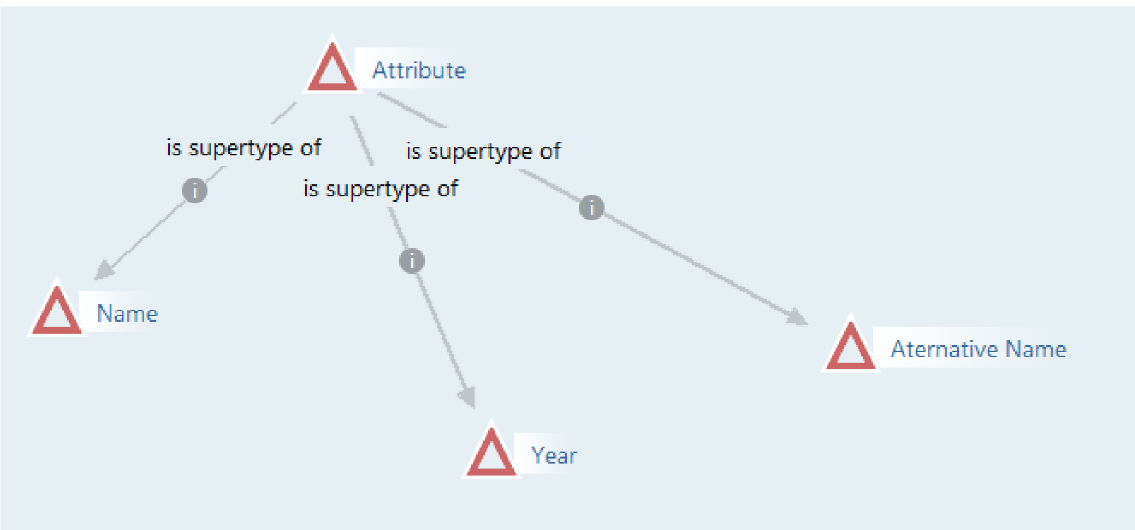
If until now we have been referring to relations in the graph editor, this concerned relation objects between specific objects of the Knowledge Graph. Moreover, the general types of relations (hence the diagrams of the relations) may also be presented in the graph editor. A relation is depicted in the graph editor as two semi-circles which represent the two directions (main direction and inverse direction). Therefore, between these two nodes there is the relation "inverse type of relation":



The presentation of a type of relation and the hierarchy within the graph editor may be shown analogue to the object editor with all supertypes and subtypes:



Attribute types may also be depicted in the graph editor — they are shown as triangular nodes.



Analogue to the type of object hierarchy the hierarchy of the relations and attributes within the graph editor may be changed by deleting and dragging the supertype relation.

1.2.5. Metamodeling and advanced constructs

1.2.5.1. Dynamic typing

Objects have exactly one type when they are created, which they do not change over their lifetime. For example, the object "George" is of the type "Person" throughout its lifetime. The type determines which properties an object can have. As a "Person", "George" can have the property "Last name", for example.

If you want to dynamically assign further types to an object, you can do this using so-called "supplemental types". The number of properties that an object can have is then also determined by the assigned supplemental types. For example, if "George" is also of the "Producer" supplemental type, it can be assigned a "produces" relationship to an object of the "Album" type.

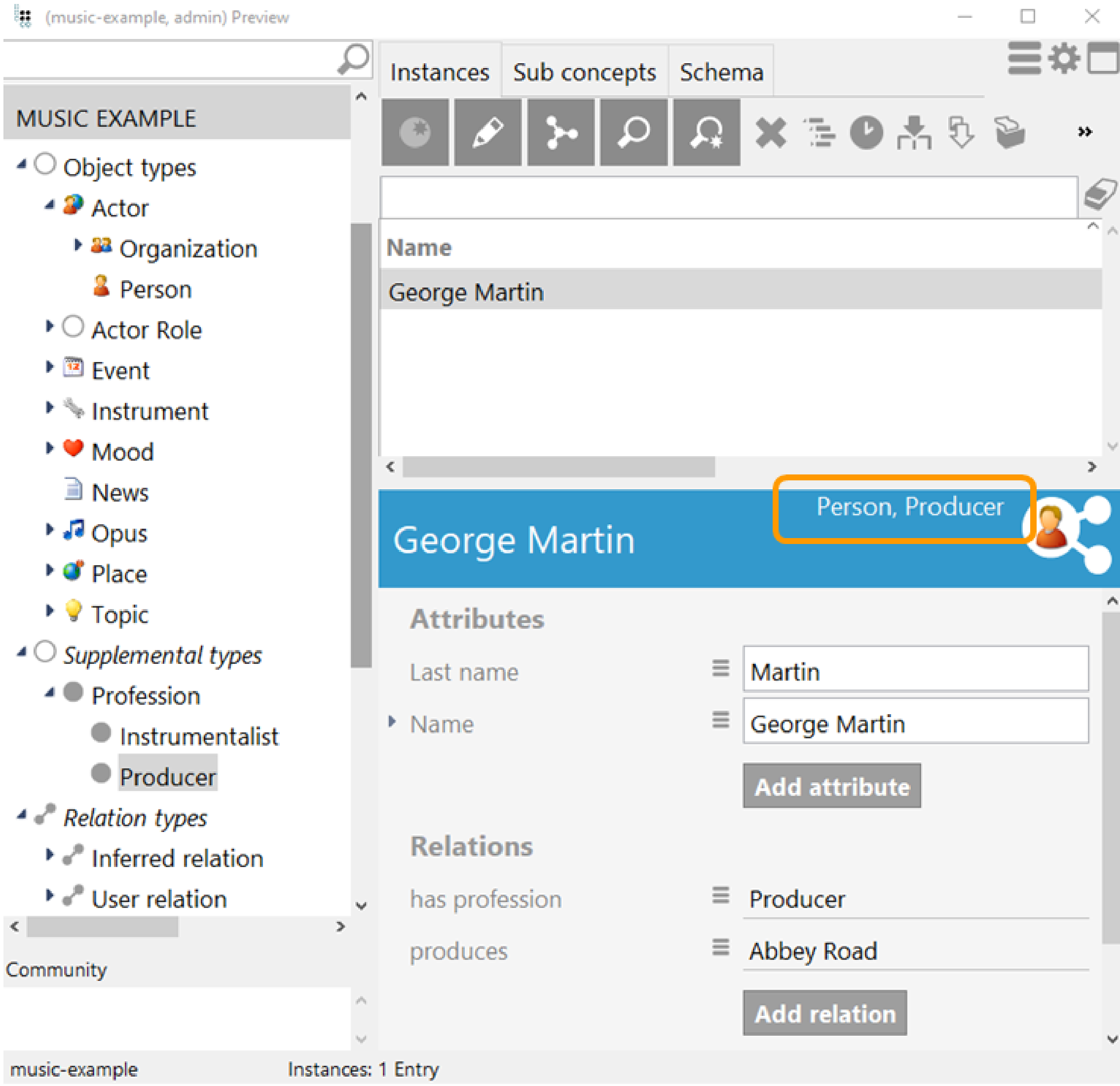


Figure 3. If supplemental types are assigned to an object, this is displayed in the title of the object view. The object is also listed in the object lists of its supplemental types.

The following steps are necessary to set up supplemental types:

1. creation of the desired supplemental types as subtypes of "Supplemental type"
2. creation of an assignment relation that allows the desired supplement types to be assigned to an object of a certain type. Assignment functionality is inherited by all subtypes of the selected extension type.

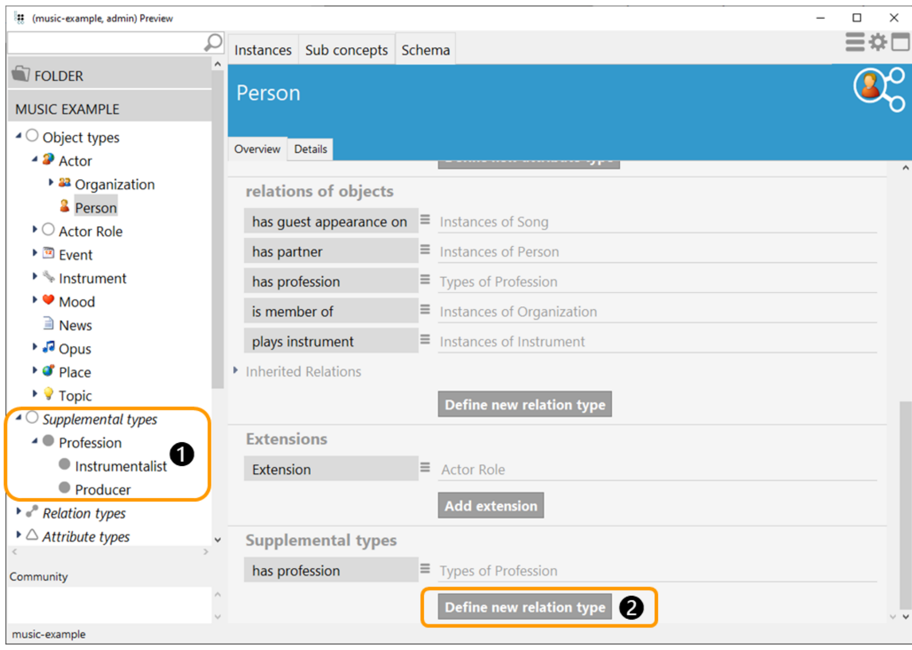
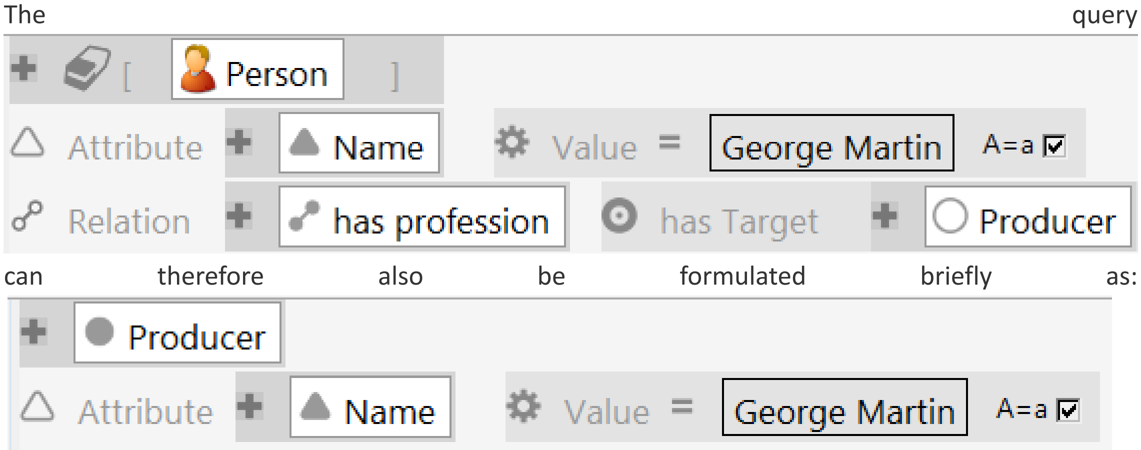


Figure 4. Steps for defining supplemental types.

Apart from its function of assigning a supplement type, the assignment relation (in the example here "has profession") is an ordinary relation that can be displayed or created in the usual way like any other relation. This applies in particular to the display in configured views or in data mappings (import and export).

In structured queries, supplement types can be used just like other types to restrict the hit list.

The query

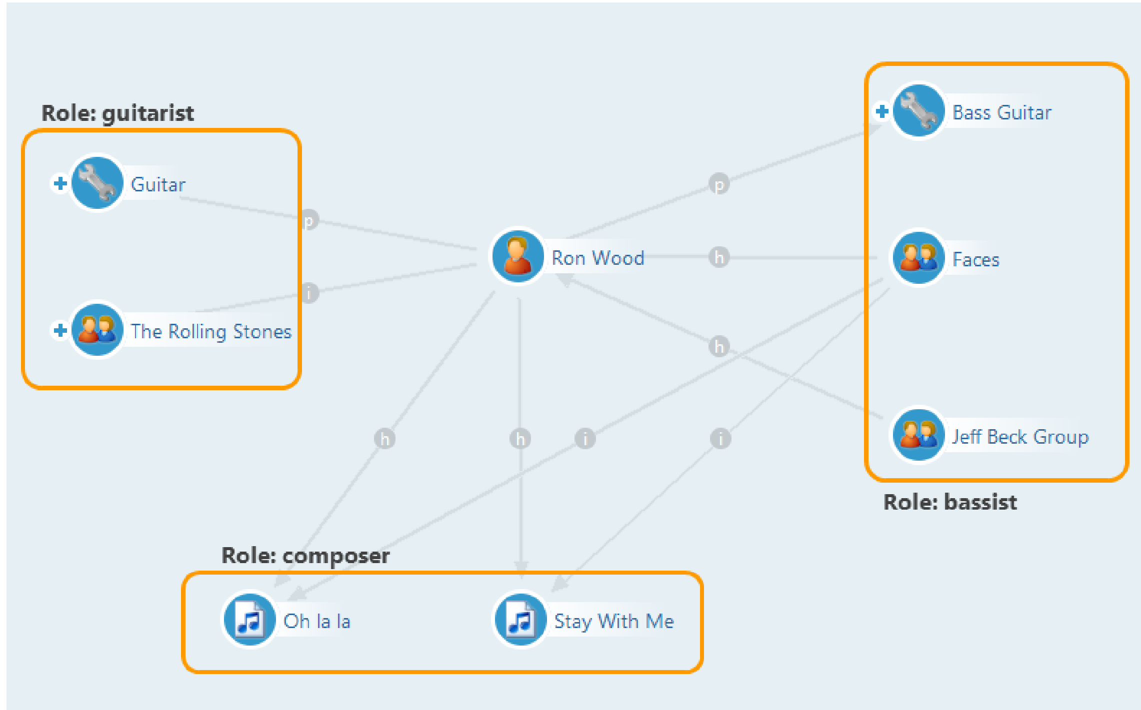


can therefore also be formulated briefly as:

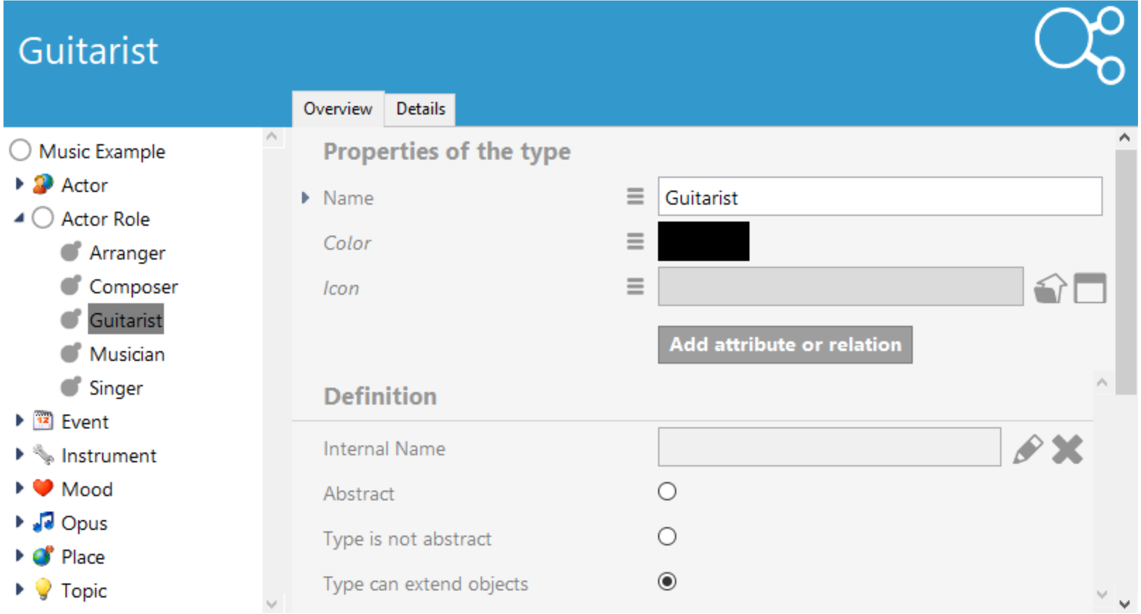
1.2.5.2. Extensions

As a further means of modelling, i-views offers the possibility of enhancing objects.

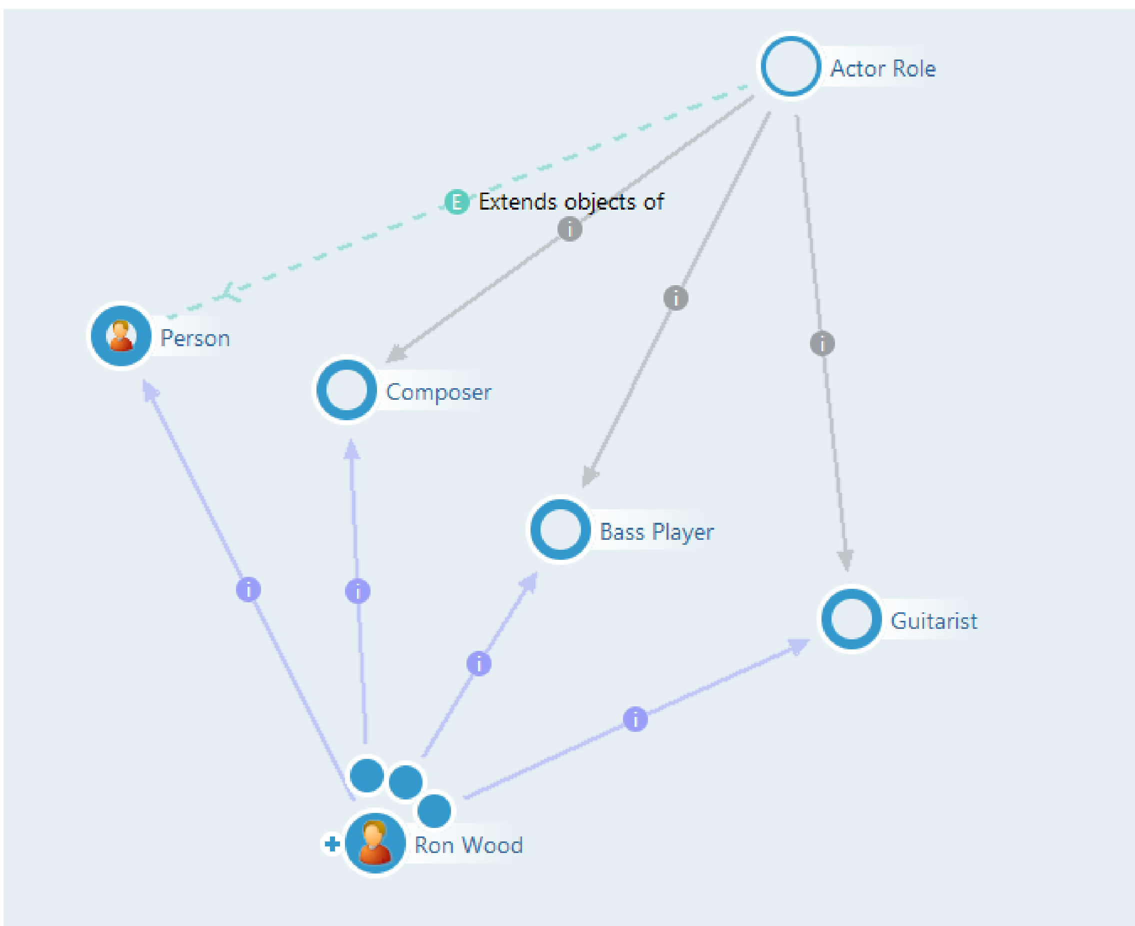
For example, if a person performs the role of a guitarist in a band but plays another kind of instrument in another band. In addition, the person exercises the role of the composer.



The fact that one person can play different roles in a Knowledge Graph may be regulated via a special form of a object type. This may not contain any objects, but enhance objects from another object type (e.g. in this case "person"). For this purpose, the object type "role" is implemented into the Knowledge Graph, for example and the different roles created for persons as subtypes: guitarist, composer, singer, bassist, etc. In order that these "role object types" may enhance objects this function will be defined in the editor for the object type by checking the box "type can extend objects":

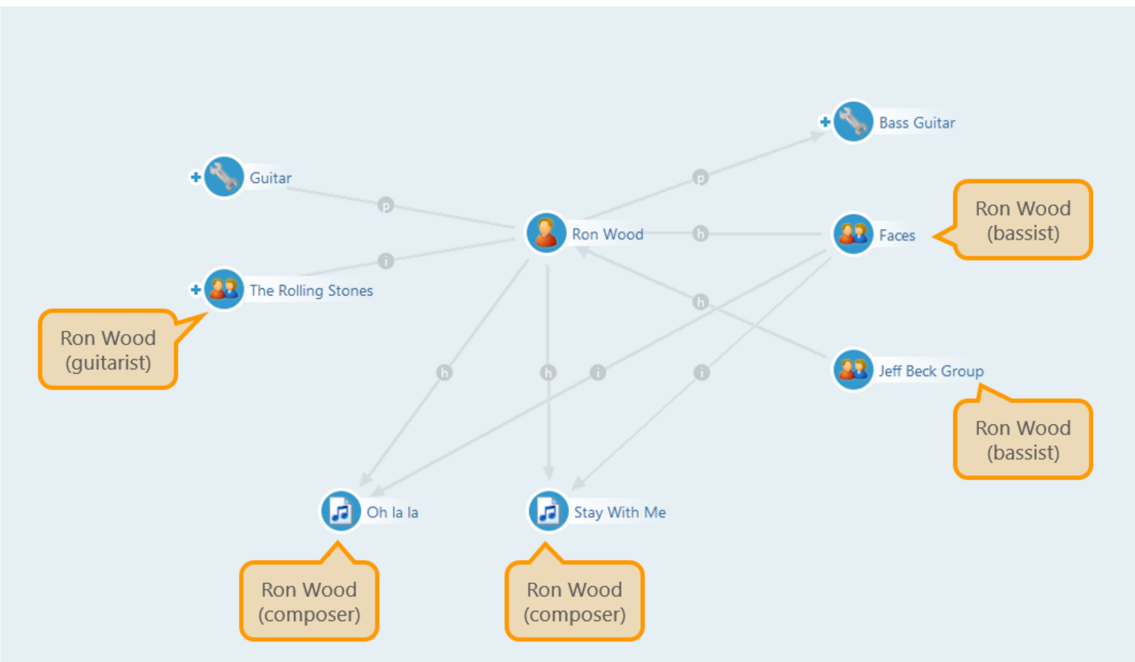


Enhancements are displayed in the graph editor as a blue dotted line:



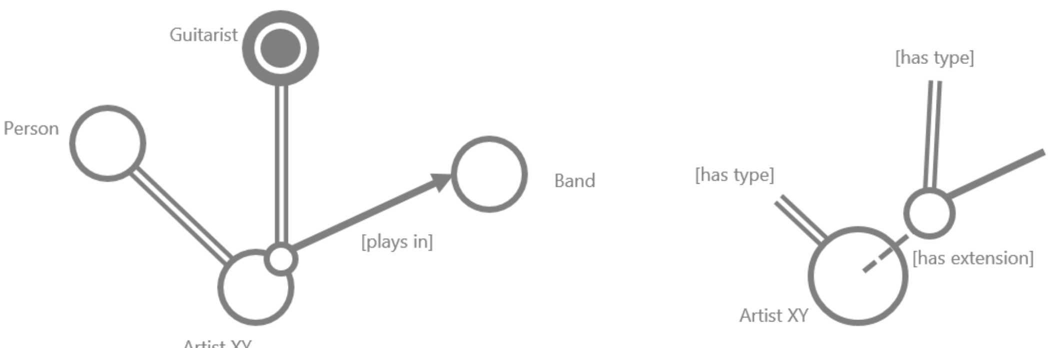
As a result of this enhancement we have achieved several things simultaneously:

- We have formed sub objects for the persons (we can also imagine these as sections or — with persons — as roles). These sub objects may be viewed and queried individually. They are not independent, when the person is deleted the enhancement "guitarist" along with the relations to the bands or titles are gone.
- We have expressed a multi-digit content. We cannot express anything on separate relations between persons, instruments, title/band — in this case the assignment would no longer succeed.



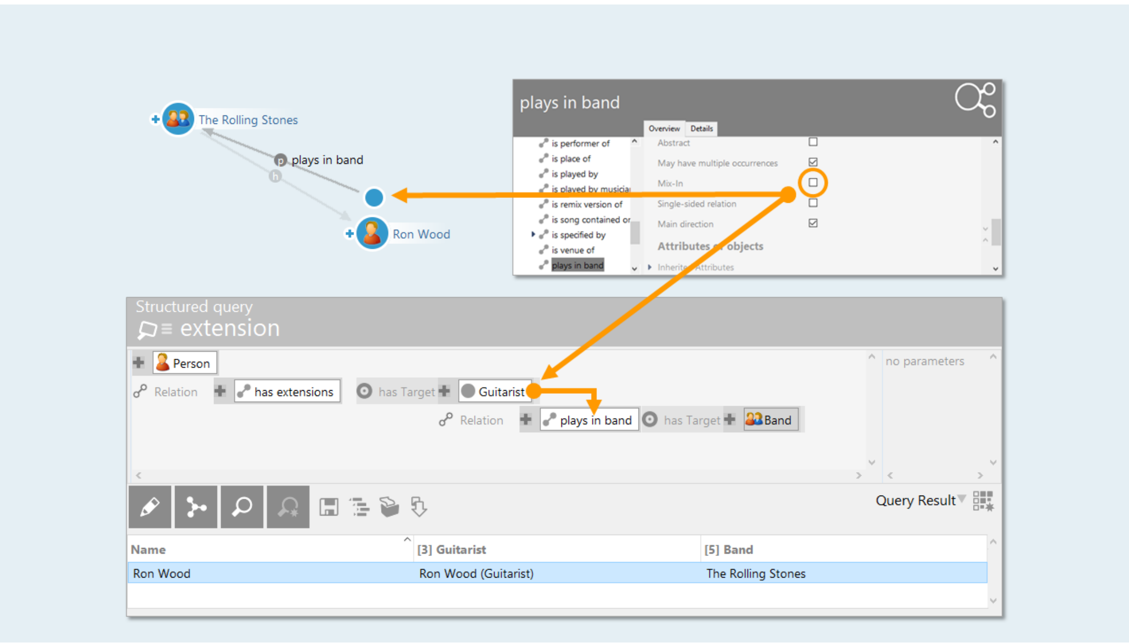
For this purpose the relation "plays in the band" for the enhancement "guitarist" has to be defined. This effect that persons inherit an additional model via the enhancement may be helpful regardless of multi-digital contents.

From a technical point of view, the enhancement is an independent object which is linked to the core individual by means of the system relation "has enhancement" or inverse "enhanced individual". Its type (system relation "has a type") forms the enhancement type.



When defining a new enhancement, two object types play a role: in our example we want to give persons an enhancement and we have to provide this information to your type "person". The enhancement itself again has an object type (usually even quite a lot of object types); in our case "guitarist". With the type "guitarist" (and with all others with which we want to enhance the persons) his specific objects will be dependent.

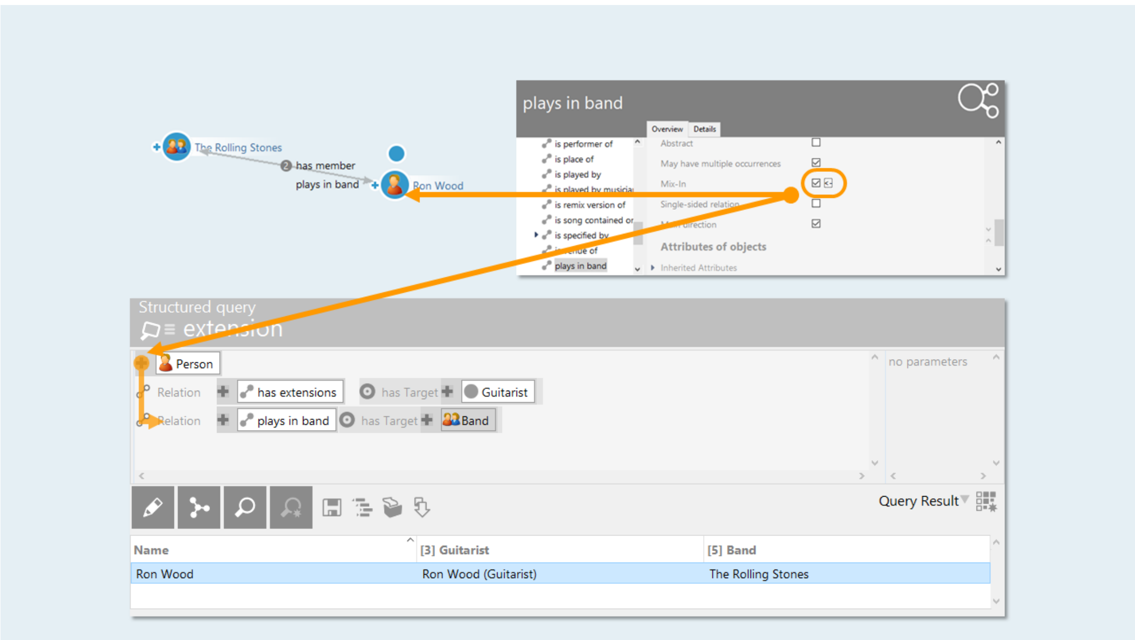
When querying enhancements in the structure search we have to traverse individual relations: From the specific person via the relation "has extension" via the enhancement object "Guitarist". From there you can reach the band via the relation "plays in band".



Mix-in

The essence of this example with the role "guitarist" is that the relation "plays in a band" is linked to the enhancement but not with the person. Hence, a consistent assignment is possible with several instruments and several bands.

If the option mix-in is selected the relation, on the other hand, is created with the core object (person) itself. The reason for this is that enhancements are sometimes not used to express more complex contents but to assign an object polyhierarchically to different types. This object inherits in this manner relations and attributes of several types.



When we setup an extensive type hierarchy of events, for example, with the subdivision into large and small events, outdoor and indoor events, sports and cultural events, we can either characterise all combinations (large outdoor concert, small indoor football tournament, etc.) or create the different types of events as possible enhancements of the objects of the type "event". Then we can assign an event via its enhancements as a football tournament and, at the same time, as an outdoor event as well as a large event. Via the enhancement "football tournament" the relation "participating team" may then be inherited, via the enhancement "outdoor event", for example, still the property "floodlight available". When we have placed these properties in mix-in they may be queried like direct properties in the events.

If a mix-in enhancement is deleted it acts like a "normal" enhancement: there has to be at least one enhancement available which entails the mix-in property. When the last of these enhancements is deleted the relation or the attribute in the core object is also deleted.

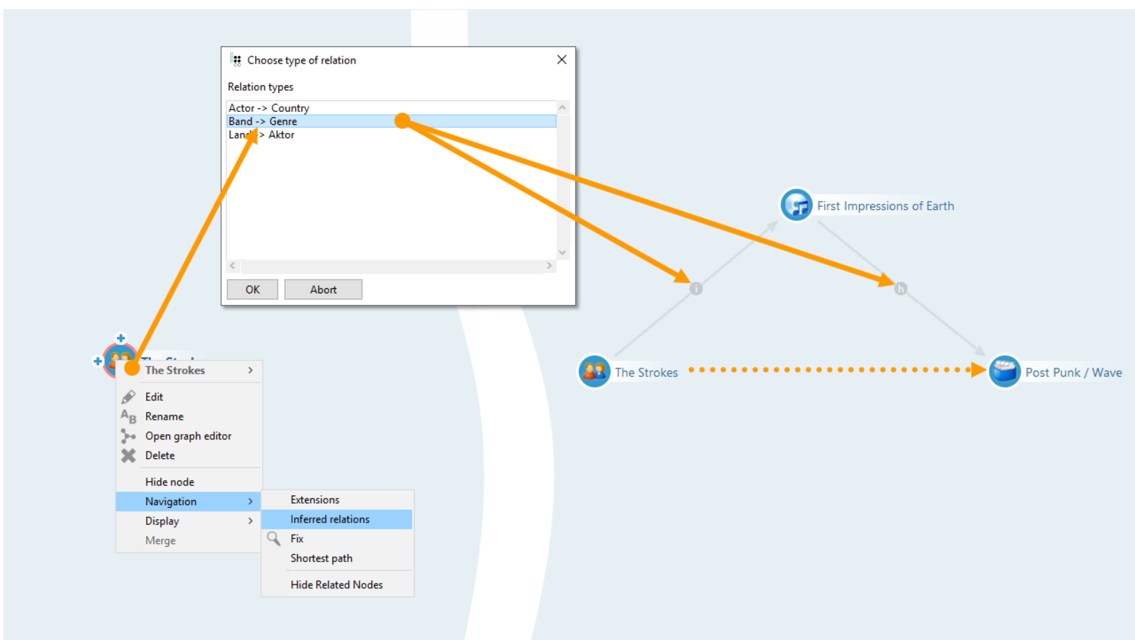
From version 6.0, extensions with mix-ins are replaced by the simpler and more flexible mechanism of supplemental types. Mix-ins can still be used, but the use of supplemental types is recommended for dynamic typing. An extension type can be converted into a supplemental type via the "Revise" menu using "Transfer extension types with mix-in properties to supplemental types". The original modeling (redundant) is retained for checking purposes, but can then be deleted without loss of data.

1.2.5.3. Inferred relations

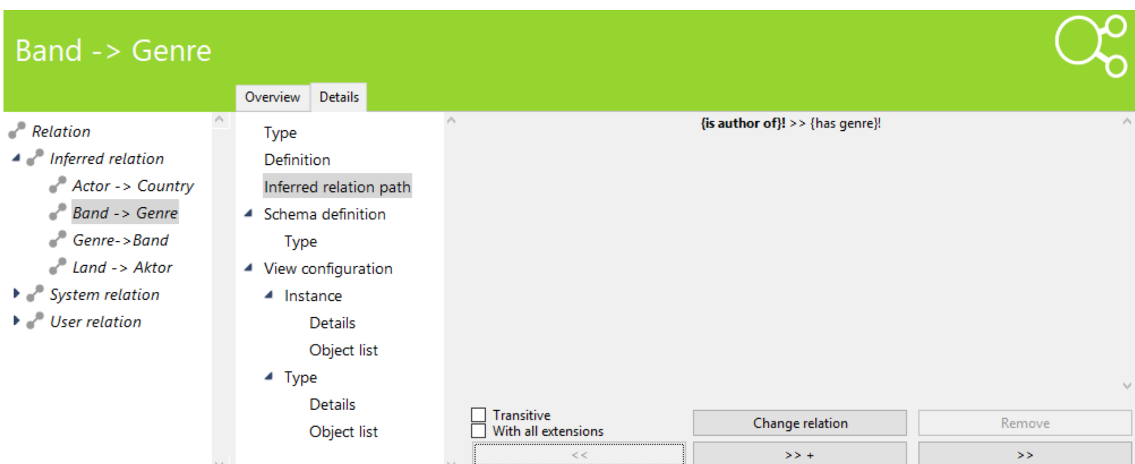
A special form of the relation is the shortcut relation. Hidden behind this is the possibility to shorten several relations already available by means of schematically predefined substitue relations.

In this manner the system can, to a certain extent, draw a direct conclusion from an object A in the Knowledge Graph which is indirectly connected to an object B via several nodes. This means that for a semantic element the inferred relation and its targets can be determined in the graph editor and in structured queries in one step.

For example, a band publishes a recording media in a certain genre of music, ergo this genre of music can likewise be assigned to this band:



In order to use inferred relations, in the form editor the inferred relation path needs to be defined via the relations "is author of" and "has genre".



Options for defining the inferred relation path:

- "Transitive": The relation may occur in any number (once to infinite).
- "With all extensions": Extensions will be included. The setting will be defined for the relation which is defined at the extension. If an inferred relation from the extension back to the core element needs to be defined, the relation type "Extends instance" must be used explicitly therefore.

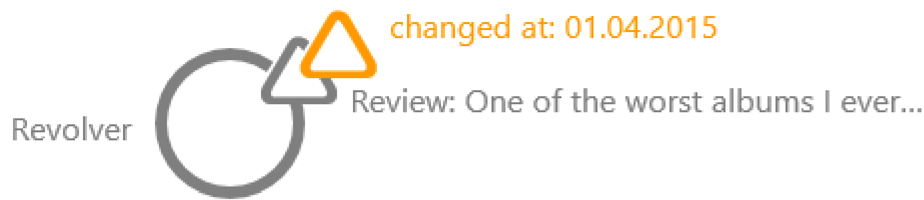
In the queries the shortcut relation can be used like any other relation as well.

In the current version of i-views it is recommended that several nodes and edges be queried via search modules as a result of the improved overview in the structured queries.

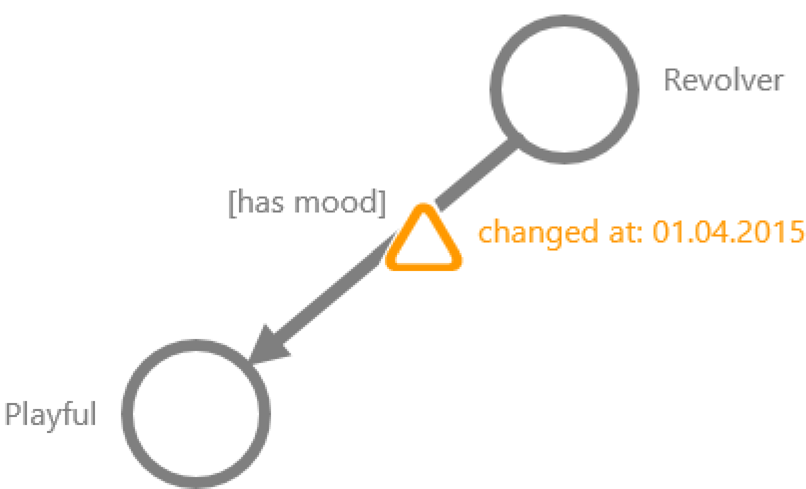
1.2.5.4. Meta properties

Up to now, properties of less complexity in object types for objects were defined. For example, users can add or edit contents to the music Knowledge Graph which we are treating here as an example via a web application. It should, however, be noted which information was changed from whom and when. To do this, attributes and relations and, in turn, for attributes and relations are required in all combinations.

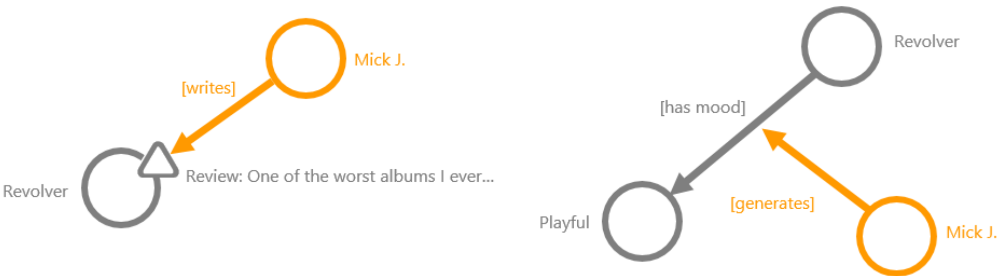
Attributes to attributes: for example, discussions and reviews are listed in the music Knowledge Graph as text attributes for music albums. If it is to be noted when discussions and reviews were added or when they were last changed we can define a date attribute which is assigned to the discussion and review attributes:



Attributes to relations: This date attribute may also be located at a relation between albums and personal sentiments such as "moods" if the users are given the possibility of tagging:



Relations may be used on attributes and on relations. For example, those users should be documented who have created or changed an attribute (e.g. review of an album) or a relation between an album and a mood at certain times:



These examples together with the editing information form a clearly demarcated meta level. Properties of properties are, however, usable for complex "primary information":

If, for example, the assignment of bands or titles to the genres be weighted, a rating as "weight" may be given to the relation as an attribute.

An attribute of a relation may also be the sum of a transfer or the duration of participation or membership.

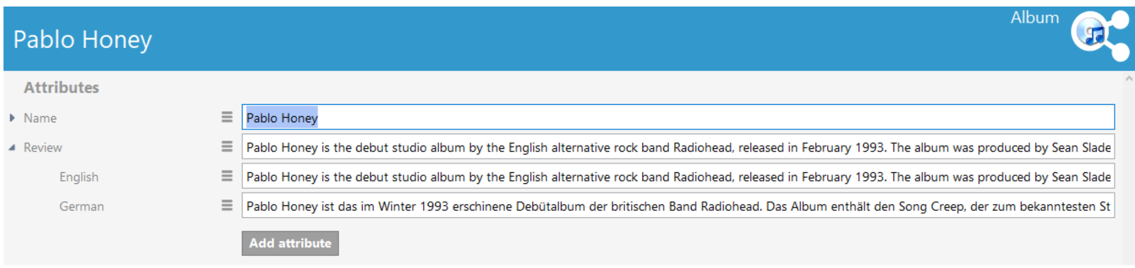
Relations to relations may also be expressed as "multi-digit contents". For example, the fact that a band performs at a festival (that is a relation) and in doing so takes a guest musician with them. He doesn't always play with the band and hence doesn't have a direct relation to it. Likewise, he cannot be generally assigned to the festival but is assigned to the performance relation.

Modelling of meta properties may, of course, also be realised by implementing additional objects. In the last example the fact that the band performed at a festival enabled an object of the type "performance" to be modelled. A significant difference is that in the meta model the primary information can simply be separated from the meta level: the graph editor does not show the meta information until it is requested and in queries, also in the definition of views the meta information can simply be left out. The second difference lies in the delete behaviour: objects are viable independently. Properties, even meta properties, are not on the other hand; when primary objects and their properties are deleted the meta properties are deleted with them.

Incidentally: properties can not only be defined for specific objects but also for the types themselves. A typical example of this is an extensive written definition with a object type, e.g. "what do we understand by a company?" That is why we are always asked whether we want to create them for concrete objects or subtypes when creating new properties.

1.2.5.5. Multilingualism

The attributes "character string", "data file attribute" and" selection" may be created multilingually. In the case of the character string attribute and data files, several character strings may then be entered for an attribute:

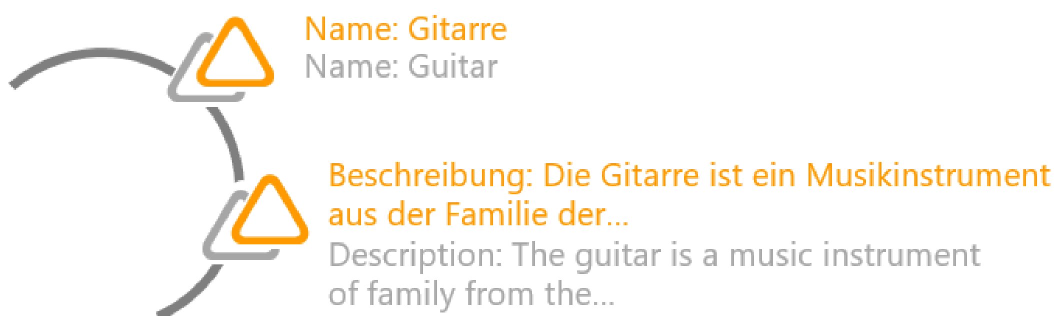


With data file attributes several images (e.g. with labels in other languages) may be uploaded analogically. In the case of selection attributes all selection options are deposited in the attribute definition; here it doesn't matter in which language the selection for the specific object is made.

All other attributes are depicted in the same manner in all languages, e.g. Boolean attributes, integers or URLs.

If the image deviates in other languages attributes adapt their image automatically, depending on the language: for example, dates according to European spelling day|month|year are shown in US format month|day|year.

In i-views separate attributes are not simply deposited for values in other languages, instead they remain as a separate layer for an attribute with language variations. You don't have to bother about the management of different languages when developing an application, but only the desired language for the respective query:



In i-views preferred alternative languages can be defined: if there is no attribute value, e.g. a descriptive text in the queried language the missing text can be shown in other languages if they are available. The order of sequence of the alternative languages may also be defined.

Multilingual settings are, for example, used in search.

1.2.5.5.1. Language Objects in i-views

In general, languages in i-views are represented by language objects. The list of languages used in the graph can be found in the *Technical* panel.

Settings related to languages, such as preferences or restrictions, are represented by relations to the respective language objects. Removing the language object accordingly results in the

dissolution of the relation and thus the removal of the associated settings.

Like other objects, language objects can be created manually as needed, but they are also generated automatically by i-views when required. Manual changes to the attributes should be avoided.

1.2.5.5.2. Language Groups and Restrictions

Individual language objects can be combined and grouped into so-called language groups. These are useful, for example, for modeling constraints such as restricting the languages available in the graph.

These constraints on individual languages or language groups can be applied in several places within the application.

For example, the view-configuration can be used to set preferences and restrictions for individual attributes, all the way up to application-wide rules for the entire Knowledge Builder.

In the schema, it is possible to individually define the set of available translations for each attribute, the corresponding language objects for each referenced language are created automatically.

1.2.5.5.3. Context dependent languages

In addition to simple languages and language groups, i-views offers three context-dependent languages for additional modeling capabilities:

- **Preferred Language**: This language represents the application’s current system language.
- **Any Language**: Suitable for modeling "any" language. For example, if the display language of an attribute is constrained to "Any language," the object will always display the first available language (in alphabetical order of ISO 639-2 codes).
- **Modeling language**: This language takes priority over all other languages. Whenever an attribute contains a translation in this language, it is always selected for display. It can be found in the list of regular languages under the code `vol`.

1.2.5.5.4. Translation selection in language search

The languages ultimately displayed/selected are derived from the set of languages remaining after all restrictions have been applied. If, in certain cases, the requested language is unavailable due to restrictions or a lack of definition, the system selects the "best possible" substitute for the requested language. This selection always follows the following order of priority:

1. **Modeling language:** If a translation for the attribute is available in the modeling language, that translation is always displayed. This applies regardless of the availability of the originally searched language.
2. **Direct match:** If the translation for the desired language is available, no further steps are necessary.

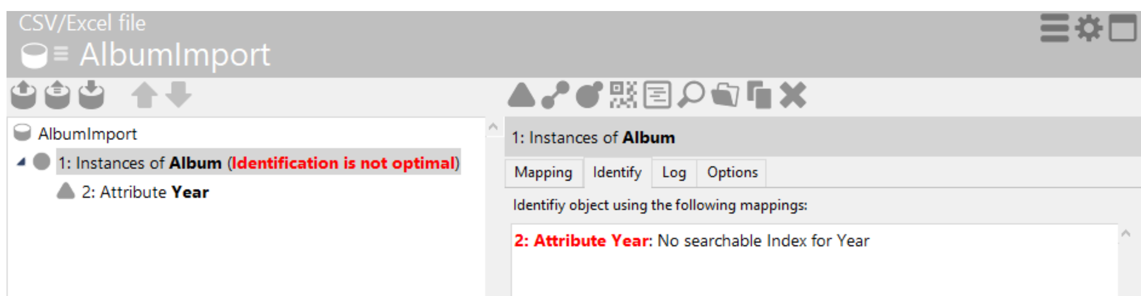
3. **Primary language:** In case the language being searched for is a localized regional variant, such as "English (US)," the system first checks whether the corresponding non-localized primary language is available, i.e., "English".
4. **Regional variant:** If this is not the case, the system will try to find an alternative regional variant for the same language, such as "English (UK)" for "English (US)".
5. **Preferred language:** Should no other option be found, a different language must be selected. The first choice here is always the "Preferred Language," i.e., the system language in which Knowledge Builder is running.
6. **Fallback list:** If no translation is available for these either, the system will sequentially check the configurable, ordered list of fallback languages. You can find it under *Settings > System > Languages*. For each item in this list, the fallback will also search for the primary language and alternative localized variants before moving on to the next one.
7. **Any Language:** If no available language has been found by this point, the system selects one of the available languages. The first language listed in alphabetical order according to ISO 639-2 is selected.

1.2.6. Indexing

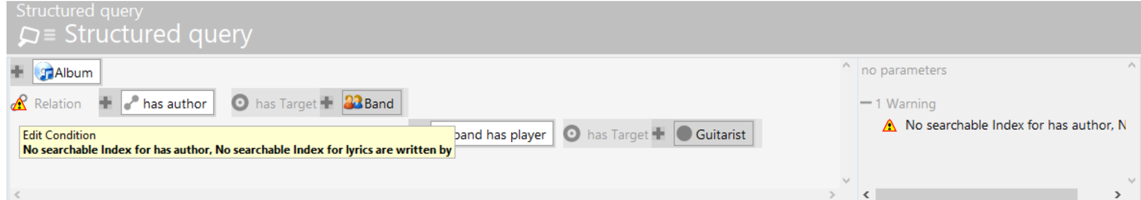
Indexing forms part of the internal data management of databases. Used correctly, the setting of indexes can improve performance significantly.

Background: In i-views, all semantic elements (types or objects) are generally stored in a cluster with their properties (attributes or relation halves). For certain transactions or uses, however, it can be better to only load part of the information. Instead of having to load the entire elements or clusters to read a few properties for queries, a corresponding index is used to refer exclusively to the required properties. Metaphorically, these indexes are both signposts and shortcuts to the required partial information.

The requirement for indexing in structured queries or during import mapping becomes apparent through various notes: In import mapping, if an object is not identified using the primary name, as expected, but through a different attribute, the note appears: "No usable index for [...]".



Import mapping with message regarding missing index



Structured query with message regarding missing index

Indexing is required for:

- Queries
- Importing data with identifying properties

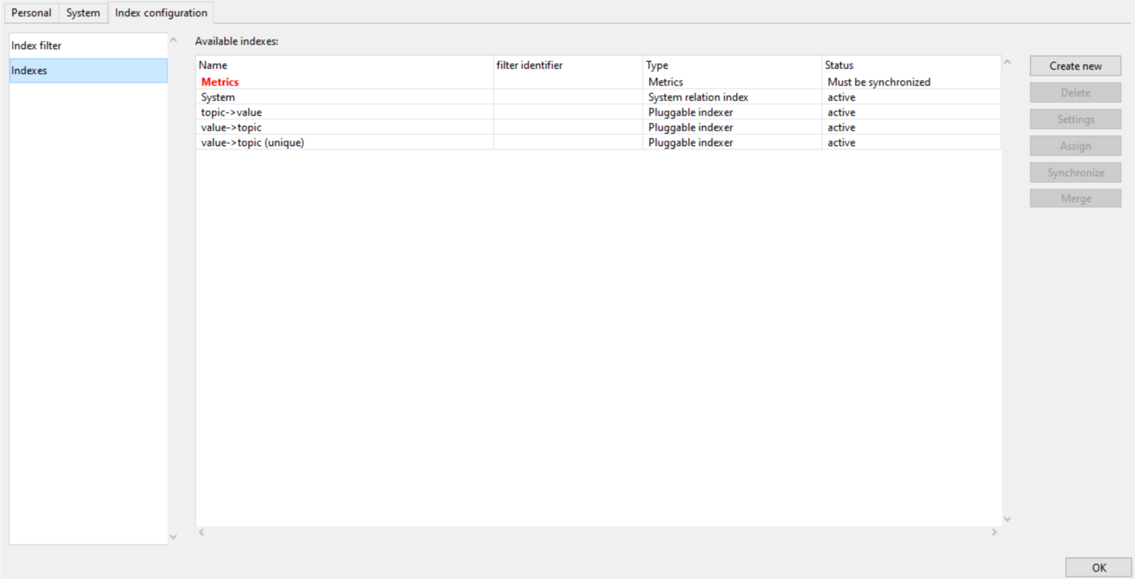
Depending on the intended use, suitable indexes must be selected for certain attributes or relations.

The indexes are **defined** in the Knowledge Builder settings. The **assignment** of the indexes can take place either in the settings of the KB or in the Detail editor of a type (Details > Indexing > Assign index).

1.2.6.1. Manage and apply available indexes

Available indexes (Settings > Index configuration)

All indexes created in the Knowledge Builder can be managed centrally in the settings.



Category "Indexes"

This setting option can be used to manage the index structures. All available index types are listed under "Available indexes". Each index type can be used for specific types of attributes or relations.

If an index is shown in grey, then the index is currently deactivated; if it is highlighted in red, then

the index is currently not synchronous.

There are buttons to generate, delete, configure, assign and synchronize on the right-hand side.

Index	Use
Lucene full text index (JNI)	Full text query
Metrics	Performance improvement in structured queries by taking into account the number of elements
System	System relations (predefined, cannot be changed) This is used for "extends object" / "has extension" / "is super-type of" / "is subtype of" relations
topic → value	To list attribute values/relation targets in object lists
topic → value (domain segmented)	To list attribute values/relation targets in object lists
value → topic and topic → value	For single-sided relations, results in a speed-up for weighted inverse single-sided relations
value → topic	Attribute values for an object
value → topic (unique)	Attribute values that may only occur once per attribute type for an object

Category "Index for relations"/ "Index for attribute value "

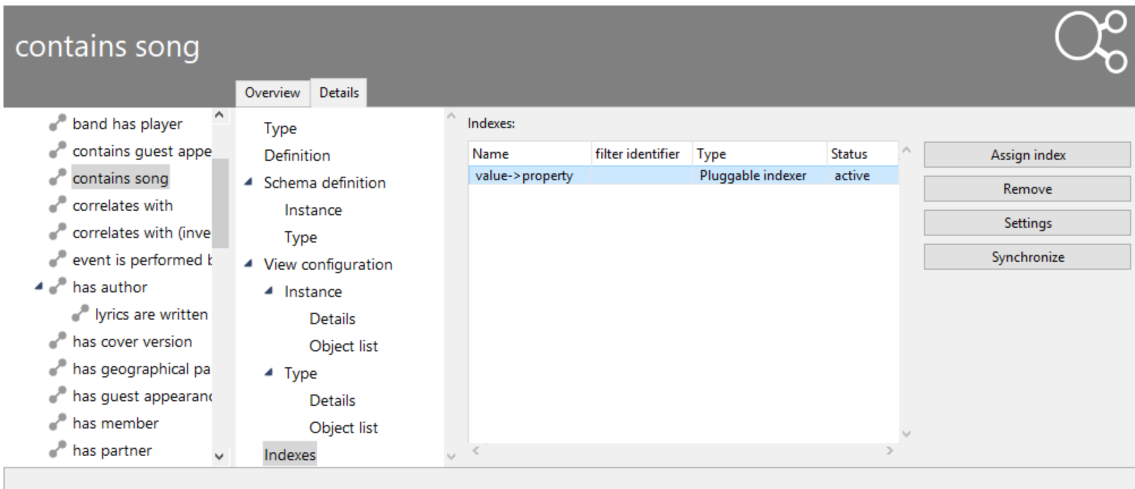
Indexes can be divided up using different aspects. First of all, a distinction can be made between forward and reverse indexes. In the case of the reverse indexes, it may make sense to refer to the property from target/value to resolve the metaconditions on the property. Ultimately, an index can optionally perform a segmentation by each type of source object in order resolve structured queries that are limited to objects of subordinate types more efficiently.

Some properties may not require an index depending on the specific application. (They can then be marked with "Ignore". They are not examined further in this optimization step.)

- Relations can use a reverse index instead of a forward index on the inverses — and vice-versa.
- Attributes can also be indexed with modified/standardized values (e.g. full text with basic word forms). A corresponding operator can then be used for search for these.

Applicable indexes (detailed configuration)

The indexes that can be used for a relation type or attribute type can be assigned using the detailed configuration.



Assigning indexes in the detailed configuration of type

Attribute types	Relation types
topic → value	topic → value
topic → value (domain segmented)	topic → value (domain segmented)
value → property	value → property
value → topic	value → topic
value → topic (unique)	

1.2.6.2. Create a new index

In the settings of Knowledge Builder, a new index can be created under: Settings > Index configuration > Indexes > Create new

The following selection is available at the start:

Index	Use
Pluggable indexer	Combined use of distributor and index modules for adapted indexing; specific configuration by means of index filters is possible
Lucene full text index (JNI)	Full text query

The following section describes the configuration of the pluggable indexers because these can be used most flexibly and cover almost all use cases.

Addable index modules

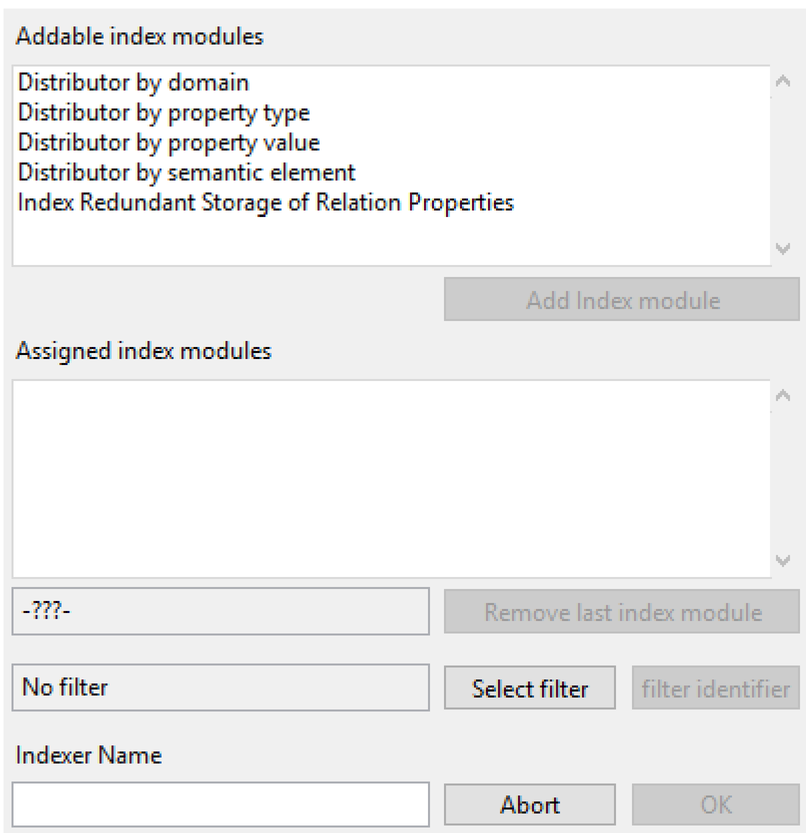
Pluggable indexers enable the administrator to create an indexer from prefabricated modules in order to achieve the corresponding indexer behavior.

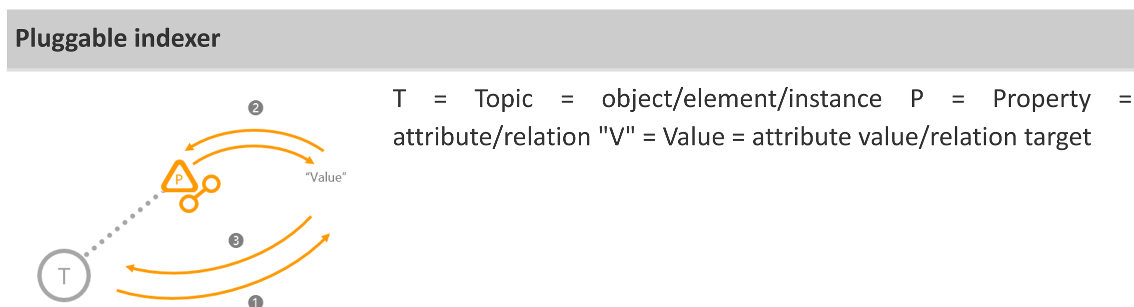
A pluggable indexer consists of distribution levels that are closed by an index level that regulates

data storage. Hence, an indexer can index both attributes and relations.

If the indexer is assigned an optional index filter, the indexer behavior can be influenced further; only suitable property types can then be assigned to the indexer.

Since properties include attributes and relations, the following section refers to an attribute value or relation target as a value of the property.





Pluggable indexer	
	T = Topic = object/element/instance P = Property = attribute/relation "V" = Value = attribute value/relation target

Distributor/index	Use
Distributor by domain (after that, all other distributors can be selected)	To search for a subset of object types that jointly use a property

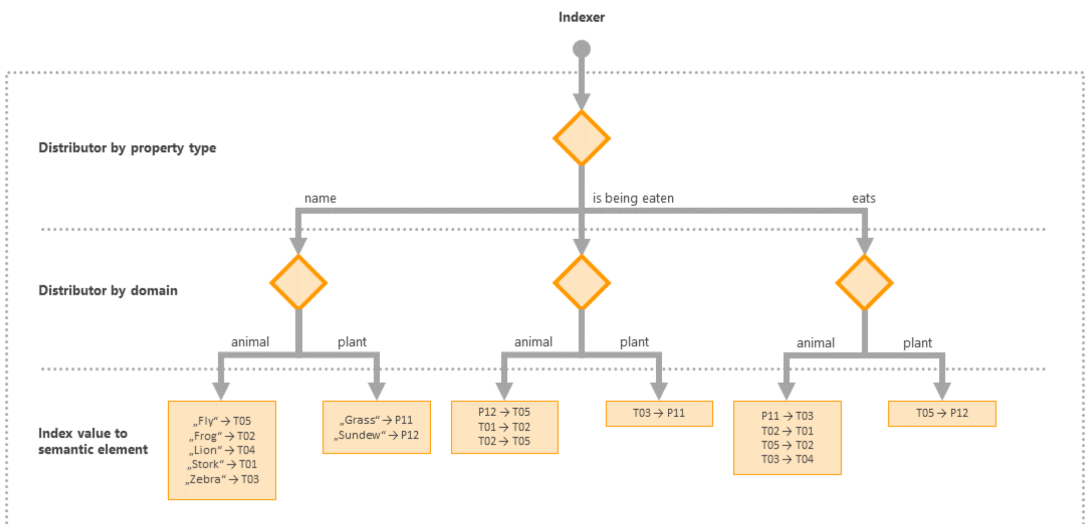
Pluggable indexer	
Distributor for each property type (index can be selected afterwards:)	Distinction between attribute and relation
Index property on value/target	Attribute → Attribute value, Relation → Target object/target type To find relation targets in structured queries with a restriction on the meta property
❶ Index object on value/target = topic → value = topic → value (domain segmented)	Object → Attribute, Abject → Target object of relation To list attribute values/relation targets in object lists
❷ Index value/target on property = value → property	Attribute value → attribute Meta-relation target → Attribute Relation target → relation Meta-attribute (value) → Relation For single-sided relations, results in a speed-up for weighted inverse one-way relations
Index value/target on property (uniqueness check)	Attribute value → Attribute To search for meta properties
❸ Index value to semantic element = value → topic	Attribute value → Attribute Relation target → Relation To support structured queries on objects with specified values/targets on attributes/relations
❸ Index value to semantic element (uniqueness check) = value → topic (unique)	Attribute value → Object (e.g.: email address)
Distributor for each property value	Together with "Index property": For compact storage of many identical values/targets; same response as for "Index value/target on property"
Distributor for each object	For single-sided inverse relations
Index redundant storage for relation properties	(Might not be used in combination with pluggable indexes) Faster display of meta properties on relations when using symmetric relational properties

Filter	
Filter type	**Use**
Latitude	For indexing an attribute type of the value type "geographical position"
Longitude	For indexing an attribute type of the value type "geographical position"
Interval start value	For indexing an attribute type of the value type "interval"
Interval stop value	For indexing an attribute type of the value type "interval"

Filter	
String filtering	.
Strings to words filter	For splitting the input string into single words

1.2.6.3. Details about indexer blocks

A distinction is made between the breakdown indexer modules and the indexing indexer modules. A breakdown indexer module partitions the index according to different aspects. Following that, there is either another breakdown or an indexing index module that stores the index entries.



The figure shows an example of how a stackable indexer consisting of three modules (without value filter) groups the index entries. This index can now efficiently provide answers to questions such as

- Which animals start with S
- Which plants either other organisms
- Which animals eat zebras (T03)
- etc.

Questions such as

- Which organisms start with S
- Which organisms eat flies (T05)

could also be answered. To do so, an indexer configuration without "Distributor by domain" would suffice (and might be more efficient depending on the data situation).

1.2.6.3.1. Distributors

- **Distributor by property type**

The most important module, without which most indexing modules cannot be added. It generally appears in first place and partitions the entries according to their property type.

- **Distributor by domain**

Enables partitioning according to the relevant terms of the property-carrying objects. The module is only useful for properties of individuals. If a property can occur in multiple object types and a search only searches for a subset of these object types, this module accelerates the search through corresponding index access.

- **Distributor by semantic element**

This module can be used for indexing to summarize the relation targets on the source object. As the previous module, it is used for mapping older indexers and its K-Infinity 3.1 only makes sense for single-sided inverse relations.

- **Distributor by property value**

Used to partition according to relation target or attribute value. In this case, only the property can still be indexed (see Index property).

1.2.6.3.2. Indices

- **Index value/target to object**

This index module is used to store an attribute value on an object or a relation target on the source of a relation in the index. This type of indexing makes sense if expert queries for objects with specified values on indexed attributes (e.g. with specified target on indexed relations) are supposed to be supported.

- **Index object to value/target**

The index module indexes in the exact opposite way as the "Index value to semantic element" and, for attributes, can be used to determine the column values of the indexed attributes for object lists. For relations, it can be used in the same way as the "Index value to semantic element" if either the inverse relation is indexed or the source object is already more restricted by the search than the target object. If you want to support expert queries with the indexed relation in both directions (source-target and target-source), the relation can be indexed either with this value and the "Index value to semantic element" or the relation and its inverse relation can both be indexed with one of the two index types. Here, it can make a difference if the index module is combined with a "Distributor by domain" because use of this distributor module for an index on the inverse relation can be used for partitioning by means of the target domain.

- **Index value/target to property** This index module is used to store values on the attribute or target on a relation in the index. This type of indexing makes sense if searches for additional

meta properties are supposed to be supported for the indexed attributes. To ensure this index can also be used in a search for the objects of the property (analogous to "Index value to semantic element"), the respective property must remain set to "Active" under "Property can be iterated" in the corresponding term editor.

- **Index property to value/target**

This index module supports expert queries to search for targets of the relations. To do so, the meta properties of the relation are used for a highly restricted process. Simple source-target conditions are not, however, supported.

- **Index property**

Together with the distributor for each property value, the same behavior can be achieved as for an index value / target to property. If there are a great many identical values or targets, this makes it possible to achieve more compact storage; otherwise, this combination has no advantages.

- **Index property value**

This index only stores the attribute values or relation targets. Using it makes sense if a "Distributor for each object" is used upstream and few objects have many values/targets.

- **Index redundant storage for relation properties**

This module can only be used by itself and is used to display the meta properties on relations more quickly if symmetric relational properties are used. No index structure is created at the technical level but the indexer can be addressed via the same configuration and programming interfaces.

1.2.6.3.3. Uniqueness check

The Index value to semantic element and Index value to property modules can be supplemented with a uniqueness check. The modules supplemented in this way are usually used for the consistency check of unique identifiers. They are available in the selection list for the addable index modules (e.g. Index value to semantic element (uniqueness check)).

If a new value is to be written and the same value is found in the index, this new value cannot be adopted. Values are recognized as identical if they are also grouped identically by all distributors of the index. If, for example, you want to perform a uniqueness check by domain only (this, for example, makes it possible for "modern" to coexist as an individual of verb and as an individual of adjective), the index must contain a Distributor by domain.

If a value filter is also configured, the uniqueness check is executed on the filtered values. This makes it possible, for example, to identify "arm" and "Arm" as identical.

NOTE	A value filter that splits strings (for full text) can be combined with the uniqueness check, but this is not usually sensible, because even a partial string can lead to duplicates after splitting, for example "The house" and "house and home".

The Index value to semantic element cannot recognize duplicate values of this property as

duplicates in an object if properties occur multiple times. It is therefore possible for two identical attributes with identical values to exist in the same object, but not in different objects. If you want to prevent this, you must deactivate multiple occurrences in the attribute term or instead use an Index value/target to property for the uniqueness check.

1.2.6.4. Details about value filter

1.2.6.4.1. Value decomposition

No atomic attribute value can be indexed for geocoordinates and interval attributes. Instead, longitude and latitude or interval start value and interval stop value are used to index one component of the value. For complete indexing, a corresponding indexer for the other component of the value must be configured respectively.

1.2.6.4.2. String manipulations

Full text filters for strings can be configured in the Admin tool. These can be used to configure which manipulation is possible on the strings, and how the strings should be split into individual words. Additional operators are then offered in expert queries, to which the respective filter label has been added, to allow a specific query to be executed using this filter.

Strings can be indexed in manipulated form by means of "string filtering", and when a query is executed, this results in all attribute values being interpreted as hits which the filter maps to the same string as the search input. By means of "string splitting", several (manipulated) sub-strings (tokens) from a text can be indexed. The related index then allows expert queries that execute a search within the string by means of the operators "Contains words" and "Contains phrase".

1.2.6.5. Metrics

An attribute "Average number (calculated)" can be created on all property types. The value of the attribute specifies how many values of the corresponding property an object from the property domain has on average.

This information enables structured queries to better decide how they determine their result set. In addition, you can create an attribute "Average number (manual)" whose value overwrites this value. (This makes sense if the domain is abstract but the property in enquiries is supposed to be used only when it actually occurs.)

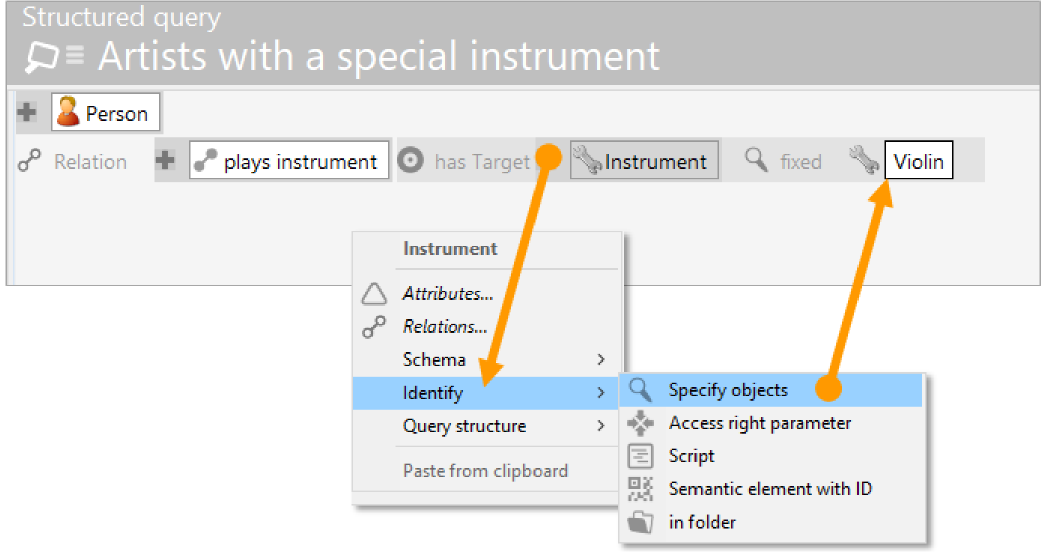
1.3. Searches / Queries

Querying of the Knowledge Graph has various subtasks for which we can configure different search modules: often we would like to process the user's entry in a search box (character strings). Usually we would like to pursue the links for the queries within the Knowledge Graph, sometimes we want to assign weights. Various types of searches are available in i-views for this purpose:

- Structured queries
- Simple/direct queries (simple search, full text search, trigram search, regular expressions, parameterised hit quality)
- Search pipelines

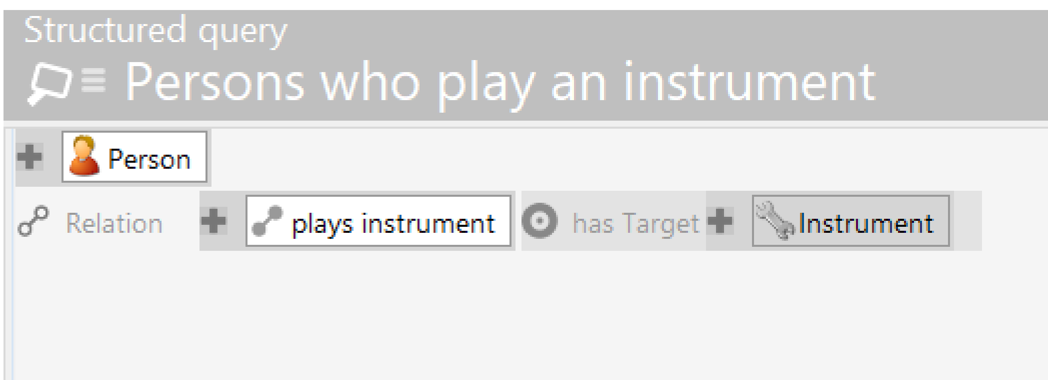
1.3.1. Structured queries

Using structured queries, you can search for objects which fulfill certain conditions. A simple example for a structured query is as follows: all persons who master a certain instrument should be found.

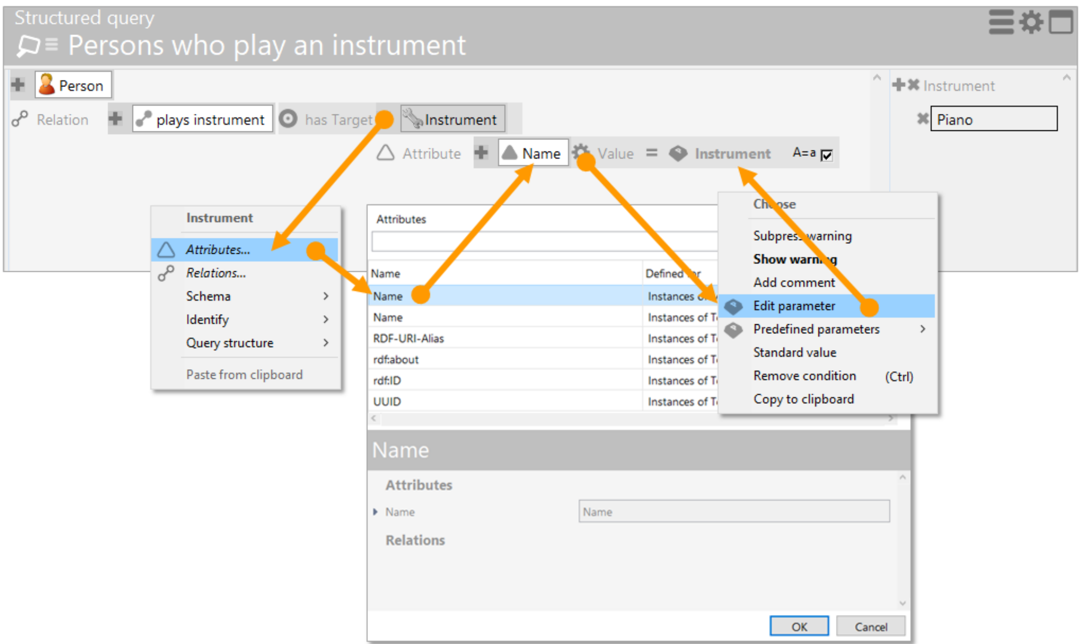


First there is the type condition: objects of the type person are searched for. The second condition: the persons have to master an instrument. Third condition: this instrument has to be the violin.

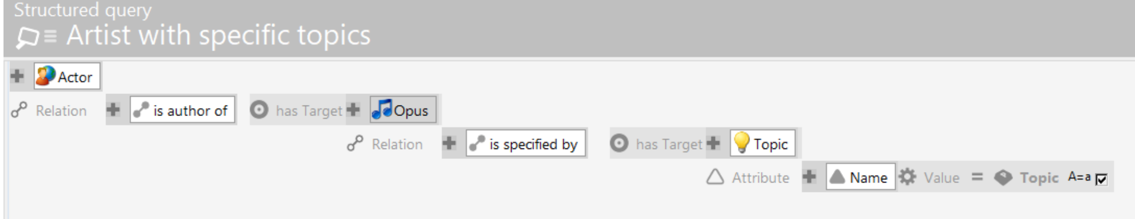
In the structured query the relation "plays instrument", the type of the target of the relation and the value of the target "violin" form three different conditions and thus also three search modes. If the third condition that the instrument has to be a violin is omitted, the query will find all persons who play at least one (arbitrary) instrument:



Often conditions (in this case the instrument) should not be predetermined or completely omitted, but given as a situational parameter in the application:



The conditions may become arbitrarily complex and can traverse the Knowledge Graph to any depth:

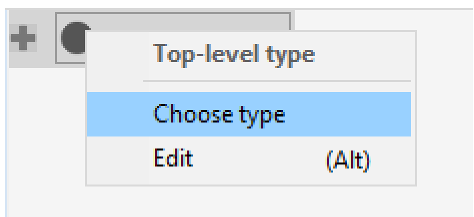


Slightly more complex example: persons or bands who deal with a certain theme in their songs (to be more exact in at least one). In this case you do not search for the name but the ID of the theme as the parameter — typical e.g. for searches which are executed via a REST service or by a script.

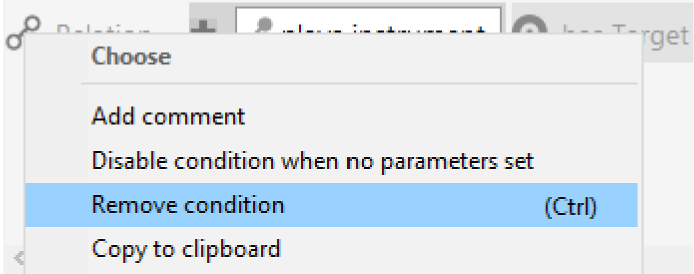
The type hierarchies are automatically taken into account in the structured queries: The type condition "Opus" in the search box above includes its subtypes "Album" and "Song". Even the relation hierarchy is taken into account: if there is a differentiation below "is author of" (e.g. "writes text" or "writes music") the two sub-relations will be included in the search. The same applies for the attribute type hierarchy.

1.3.1.1. Interaction

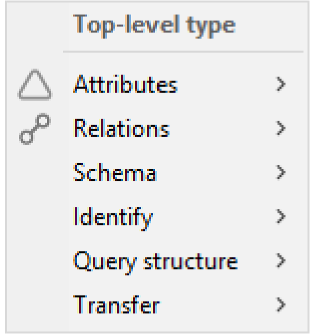
When a new structured query is created, the root condition contains the root object type of the Knowledge Graph by default. In order narrow down the query you can simply overwrite the name or select *Choose type* by clicking on the icon.



The button allows you to add more conditions to the structured query. Deleting conditions takes place at the beginning of each line where the type condition is listed (relation, attribute, target, etc.). Alternatively, conditions can be removed by using the shortcut Ctrl + Click.



When you click on the button the following menu will appear which may vary slightly depending on the context.



A complete explanation of all conditions and options of the structured queries can be found in the next chapters.

1.3.1.2. Use of structured queries

One of the main purposes of structured queries is to provide information on a certain context in applications. The structured query from the last section, for example, can enable end users in a music portal to generate a list of all artists or bands who cover subjects such as love, drugs, violence etc. in their songs.

To do so, the structured query is usually integrated into a *REST service* via the query's *registration key*. We include the subject in which the user is interested as a parameter in the query with the user's ID.

> A user enters a search string to search for their topic. Hence, there is no ID but only a string that is to be used to identify the topic. However, the query result is supposed to show immediately which bands have written songs on the subject. For this purpose, a structured query can be integrated into a *search pipeline* as a component, after the query that processes the search string.

One of the reasons why structured queries are such a central tool for i-views is that the conditions for *rights and triggers* are defined with structured queries. Let's assume the only people allowed to leave comments in a music portal are artists and bands. In the rights system, you can thus specify that only artists and bands that have written at least one song on a topic may leave comments on this topic. Structured queries can also be used in exports to determine which objects are to be exported.

All these uses have one thing in common: we are only interested in qualitative, not weighted statements. This is the domain of structured queries in contrast to search pipelines.

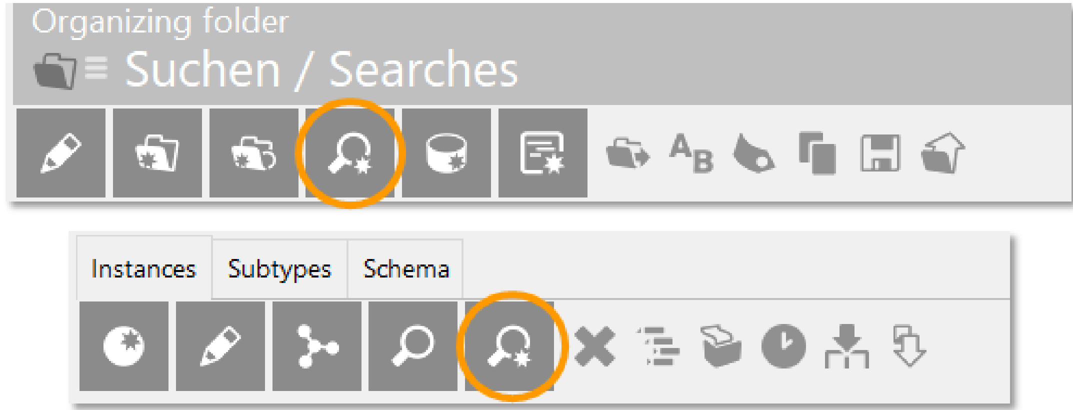
Last but not least, structured queries are also important tools for us as knowledge engineers. We can use them to get an overview of the Knowledge Graph and compile *reports* and *to-do lists*. Here are some examples of questions that can be answered using structured queries:

- Which topic is featured by many artists/bands?
- Do specific topics have to be removed because too many relations have amassed or conversely should rarely used topics be merged or closed?

For ease of use, it makes sense to be able to organize structured queries in *folders*.

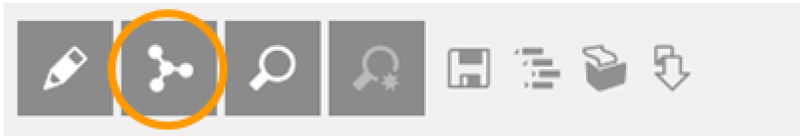
1.3.1.2.1. Implement

The structured queries are implemented in the organizing folder tab or on the results tab by means of the button *New query*:



The search results can then be further processed (e.g. copied into a new folder) but they are not kept there permanently.

The path which the structured query has taken may only be viewed in the graph editor to backtrack it. To this end, one or more hits are selected and displayed using the button graph.



A structured query may be copied in order to create different versions, for example. Likewise there is the possibility of saving them in XML format, regardless of the Knowledge Graph. The structured query may therefore be imported into another Knowledge Graph. However, this is limited to versions of the same Knowledge Graph, e.g. to backup copies, because the structured query references types of objects, relations and attributes via their internal IDs.

1.3.1.3. Structure of structured queries

With structured queries we are able to express indirect conditions: you may arbitrarily traverse between the elements throughout the structure of the Knowledge Graph. Artists and bands may be found who wrote songs on certain topics but which we cannot name specifically using their titles.

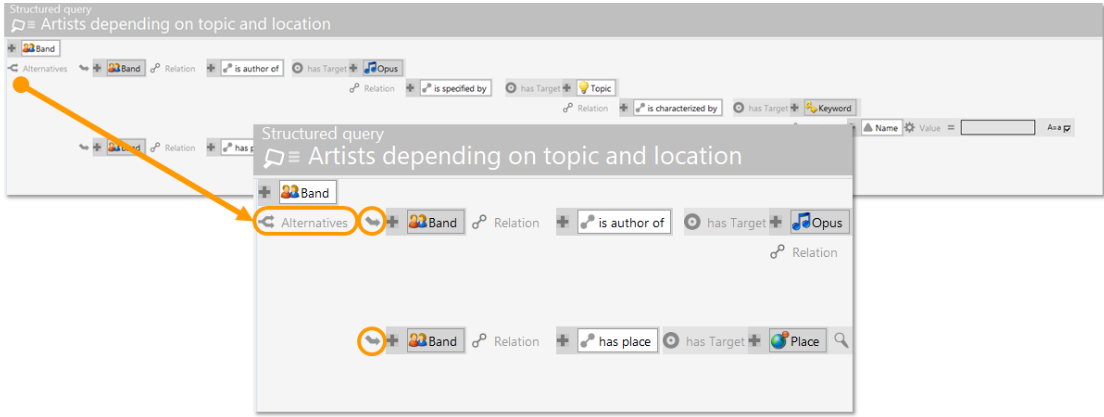
1.3.1.3.1. Multiple conditions

Condition chains may be arbitrarily deep, but it’s also possible to express several parallel conditions. Additional conditions are added to any condition element as a further branch:

Several conditions: English bands with songs on a certain subject

1.3.1.3.2. Alternative Conditions

In the example mentioned above only artists or bands can be found who created songs on a defined subject and who come from England. If, instead, we want to find all artists and bands which fulfil one of the two conditions they will be expressed as *alternative*. By clicking the symbol of the condition in the form of the relation "is the author of" you can select an alternative from the menu:



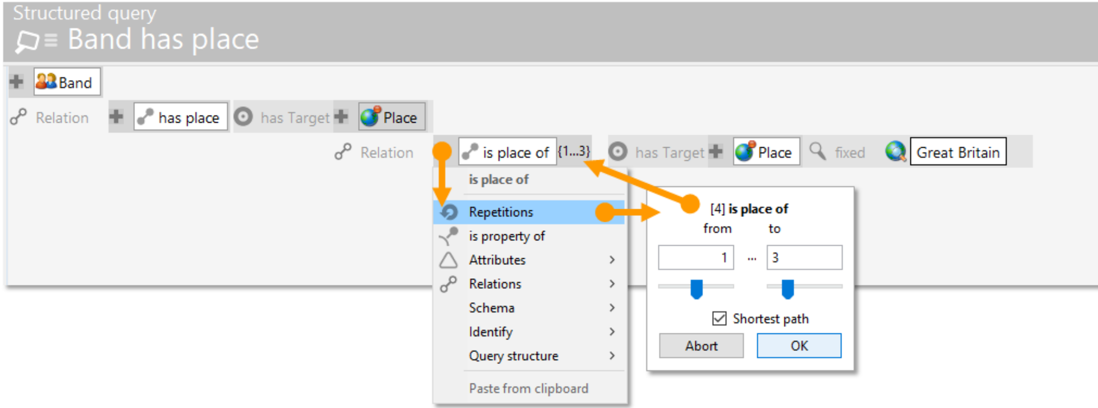
Alternative conditions: the band either has to be English or have songs on a certain subject

If there are further conditions outside the alternative bracket there are objects in the hit list which fulfil *at least one* of the alternatives and *all other* conditions.

1.3.1.3.3. Transitivity / Repetitions

Let’s assume the bands are assigned to either cities or countries within the Knowledge Graph. Of these, in turn, it is known which cities are in which countries. In order to document these contents in the search it is possible to expand the condition string: we are able, for example, to search for bands which are assigned to a city which, on the other hand is in England. However, in this manner those bands will not be found which are directly assigned to England. In order to avoid this we can state in the relation "is located in" that it is optional and therefore is not required to exist.

Simultaneously, we can also include hierarchies which are several levels deep using the function *Repetitions*. For example, the band ZZ Top is known to originate from the city of Houston which is in Texas. In order to also retain the band as a result when bands from the USA are queried we can state in the relation "is located in" that this relation has to be followed up until n repetitions are reached:



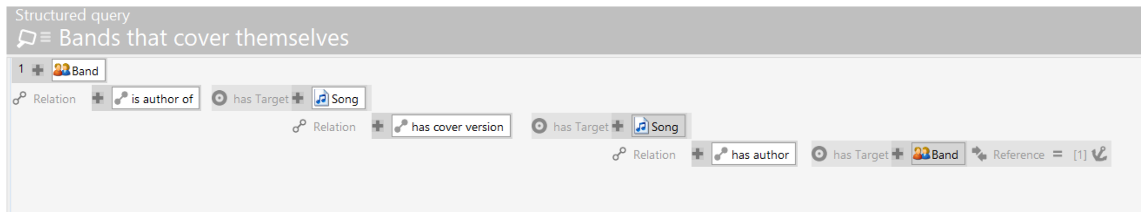
1.3.1.3.4. Negated conditions

Conditions can be negated. For example, if punk bands are searched for, which do not come from Great Britain. To this end, the negative condition is setup as a so-called *Utility query*:

The utility query retrieves bands from Great Britain. From the main search a reference to the utility query can be established specifying that the search results must not satisfy the criteria of the utility query. In this manner we remove the results of the utility query from those of the main query and only obtain bands which do not come from England.

1.3.1.3.5. References

References allow referring to other conditions of the same query:



Here the last condition references the first one, i.e. the band who writes the cover version also has to be the author of the original. Without a reference the search would read as follows: bands which

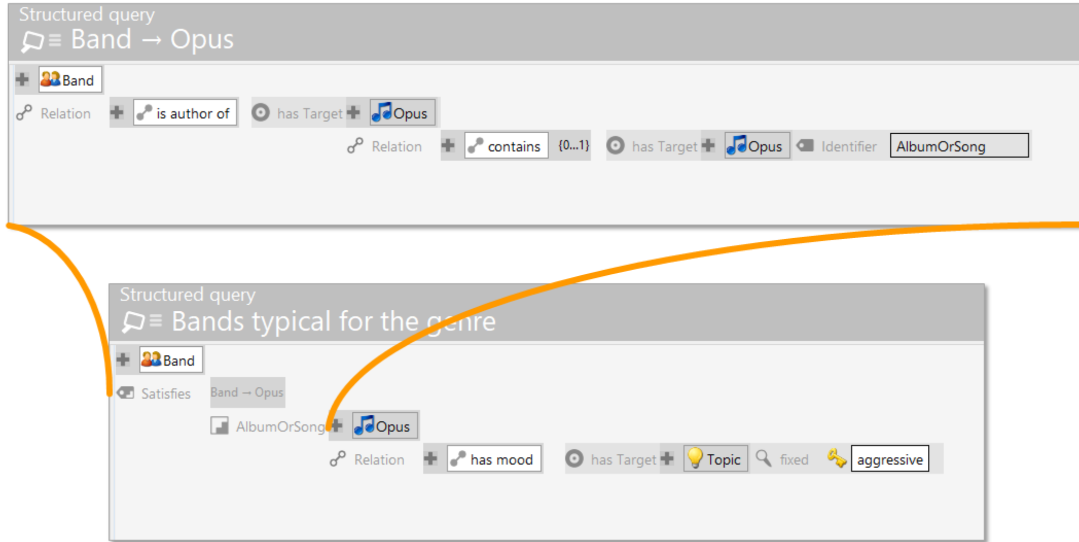
have written songs which cover other songs which were written by any (other) bands.

1.3.1.3.6. Other options in building the structured queries

Structured query macros

Other structured queries, or more generally other searches of any kind, can be integrated into structured queries as macros. In doing so, there is the possibility of outsourcing repeating, partial queries into your own macros and thus, for example, to adapt the behaviour at a central location when changing the model. A macro can be integrated into each condition line.

An example from our music graph: The works of a band include singles, albums and the songs included in these albums. We can reuse this partial query, for example in a structured query which returns the bands for a certain mood. We start this query with a type condition — we are looking for bands — and integrate the pre-defined macro as a condition for these bands:



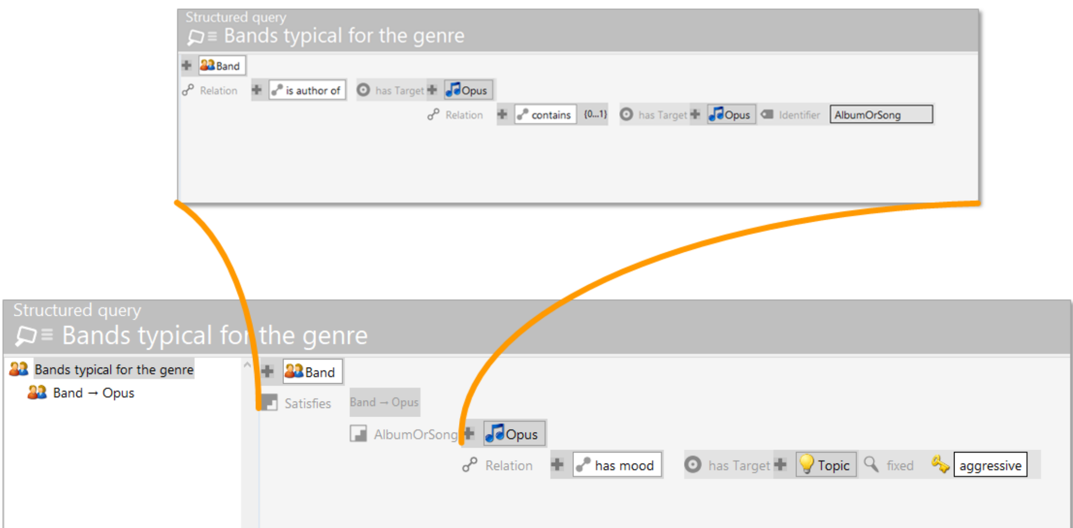
The objects which return those which are integrated into the structured query as macros have to match type determined by the condition to which they are linked.

With the aid of the *identifier* tool, the query can still be continued with additional conditions.

In our case the albums and songs from where the macro query originates are defined by the invoking query: Namely albums and songs with the mood "aggressive". Integrating the search macro into the structured query is carried out through the menu *Query structure*. Under *Structured query macro (registered)* there is a list with all the registered macros. The advantage is that the macro can be reused for another structured queries.

WARNING	As soon as the macro is deregistered, it is deleted and not available for other queries anymore.

It is also possible to use local macros for structured queries. In this case, the macro doesn't get a registry key and is only accessible for and within the respective structured query.



Query

Using the search condition *Query*, the results of a simple search or a search pipeline may serve as input for a structured query. The input box contains the search entry for the simple search. Further conditions can enable a simple search to be filtered further, for example.

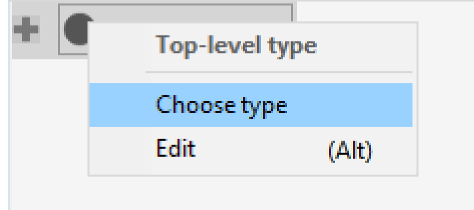
Cardinality condition

A search for attributes or relations may be expanded with a cardinality operator (characterised by a hash tag #). You may use the cardinality greater than or equal to, less than or equal to and equal. The normal equal operator of the relation or attribute condition corresponds to greater than or equal to 1.

1.3.1.4. Conditions in detail

1.3.1.4.1. Type condition

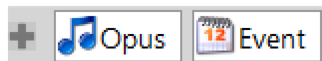
The root of the structured query determines which objects should appear as the results. To do so, click on the type icon for the first condition and select *Choose type* in the menu, the input mask then starts in which the name of the type can be entered.



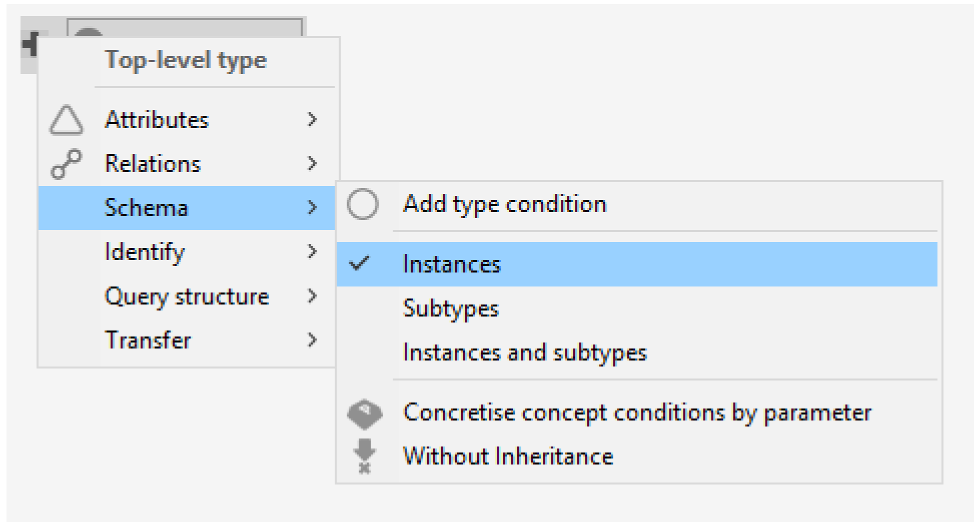
Alternatively, you can simply overwrite the text after the type icon with the name of the type.

In the second step the relation condition is added. For example, a search is made for the place of origin of a band and "has place" is set as a relation condition. The target type of the relation is added automatically which, however, can also be changed (if, for example, the "has location" relation for countries, cities and regions applies but we only wish to have the cities).

Multiple types in a single type condition are interpreted in terms of an "or" logic in the query. For example, we search for works or events on a particular style of music as follows:



There are further functions available for a type condition. In the context menu accessible through the button, there is the *Schema* item:



We can just search for types of objects instead of specific objects or both at the same time by checking the item *Subtypes* or *Instances and subtypes* in the *Schema* submenu.

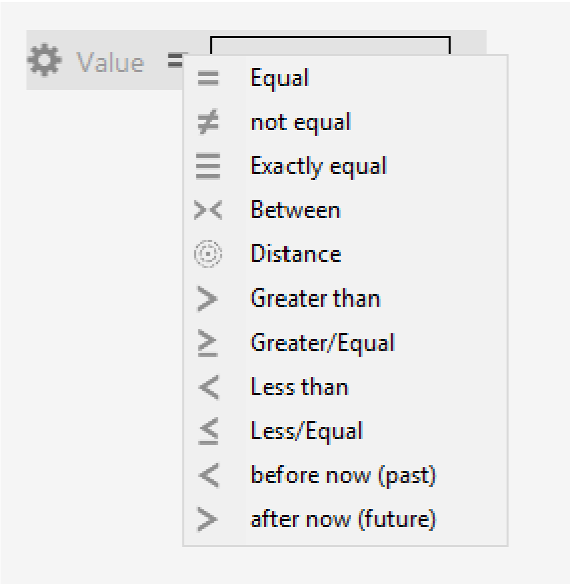
This is what the condition looks like when a search is made for both specific opus as well as subtypes of opus (albums and songs).

Without inheritance

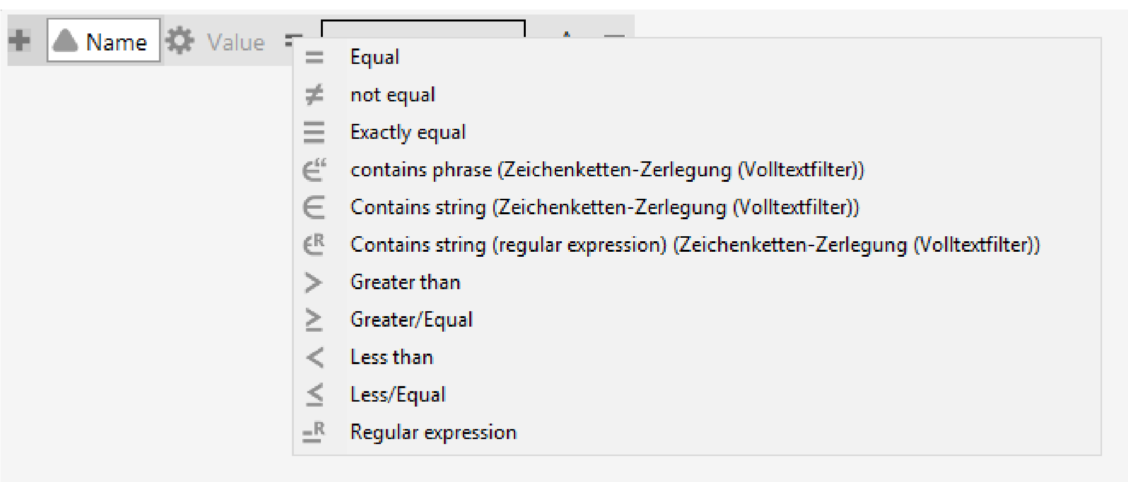
Normally, inheritance is automatically taken into consideration for all types of conditions of the structured query. If a search is made for events in which bands play a certain style of music, all subtypes of events are then incorporated into the search and then we are provided with indoor concerts, club concerts, festivals, etc. In the vast majority of cases this is exactly what is desired. For exceptions there is the possibility of switching off the inheritance and restrict the search to direct objects of the type event, i.e. by excluding the subtypes of objects.

1.3.1.4.2. Operators for the comparison of attribute values

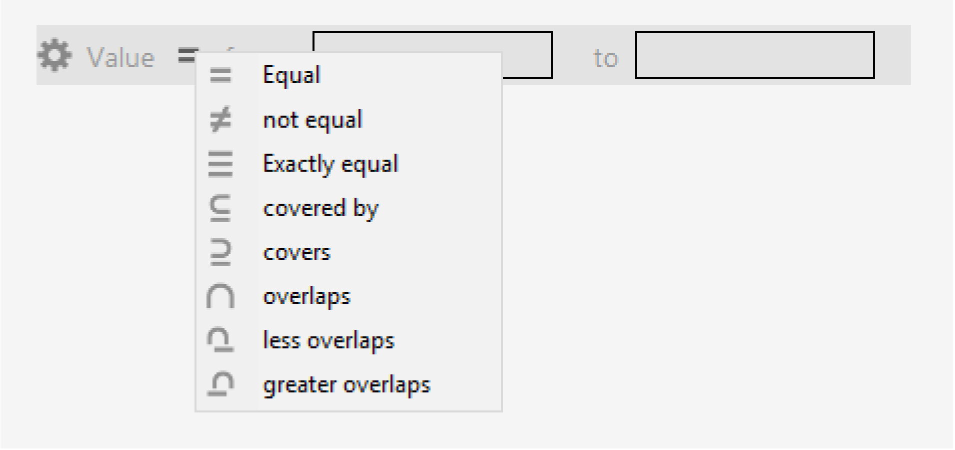
For attributes it is possible to specify value conditions, e.g. when searching for bands which were founded after 2005 or songs which are more or less 3 minutes long or songs which contain the word "planet" in their title. These require comparison operators. The type of comparison operators which i-views offers us depends on the data type of the attribute:



Comparison operators for dates and quantities



Comparison operators for character strings



Comparison operators for intervals

Operator overview

Operator	Attribute value types	Description	Example
Equal	Any attribute value typ	Identigfies values that are equal to the searched value.In case of character strings, wildcard characters "*" (arbitrary strings) and "?" (one arbitrary character) are supported.	Search term "Star*" finds "Star" and "Start"
Exactly equal	Character String	Identifies character strings which are identical with the search term without using wildcard characters.	Search term "Star*" finds "Star*", but not "Star" or "Start"
Contains phrase	Character string with full text index	Identifies character strings which contain the searched terms in forms of a subset.	Search term "Farmer George" finds "Farmer George Green", but not "George Farmer"
Contains character string (strings to word filter)	Character string with full text index	Identifies character strings which contain all words of the searched term in an arbitrary order	Search term "Farmer George" finds "Farmer George Green" and "George Farmer", but not "George Grey"

Operator	Attribute value types	Description	Example
Contains character string (regular expression)	Character string with full text index	Indentifies character strings of which at least on word matches the search terms derived from the regular expression.	Search term "Ba[yi]e?r" finds "Silke Bayer" and "Emil Bair", but not "Bauer"
Regular expression	Character string	Identifies character strings which match the search terms derived from the regular expression.	"\d+\s\w+" finds "64293 Darmstadt"
Not equal	Any attribute value type	Identifies values which are not equal to the searched value. In case of character strings, wildcard characters "*" (arbitrary strings) and "?" (one arbitrary character) are supported.	
Greater than	All attributes with assortable values	Identifies values (and hence the elements carrying the attribute) which are greater than the searched values.	
Greater/Equal	All attributes with assortable values	Identifies values greater than or equal to the searched value.	
Less than	All attributes with assortable values	Identifies values less than the searched term.	
Less/Equal	All attributes with assortable values	Indentifies values less than or equal to the searched value.	
Equal (geo)	Geo		
Equal now (present)	Date		
Before now (past)	Date	Identifies date values that are situated in the past.	

Operator	Attribute value types	Description	Example
After now (future)	Date	Identifies date values that are situated in the future.	
Distance	Date, Geo, Number	Identifies values whose distance to the searched value equal to the maximum of the given distance value (date: number of days, geo: distance in meter). Parameter values require a tilde character to separate the target value from the distance, e.g. "2019/10/01 ~ 30" — i.e. on 2019/10/01 plus/minus 30 days.	Search value "2019/10/01" with distance 30 will return the result "2019/10/15", but not "2019/11/01"
Between	Interval	Identifies intervals which completely comprise the searched value. Parameter values require a hyphen character, e.g. "10.1.2005 — 20.1.2005".	Searched value "1 — 5" returns "1 — 3", but not "3 — 6"
Covers	Interval	Identifies intervals which comprise a common, non-empty partial interval with the searched value.	"2 — 4" finds "1 — 3" and "3 — 6", but not "4 — 5"
Covered by	Interval		
Overlaps	Interval	Identifies intervals which share a common, non-empty partial interval with the search value.	"2 — 4" finds "1 — 3" and "3 — 6", but not "4 — 5"

Operator	Attribute value types	Description	Example
Greater overlaps	Interval	Identifies intervals which share a common, non-empty partial interval, containing the lower limit of the search value interval.	"2 — 4" finds "3 — 6", but not "1 — 3"
Less overlaps	Interval	Identifies intervals which share a common, non-empty partial interval, containing the upper limit of the search value interval.	"2 — 4" finds "1 — 3", but not "3 — 6"

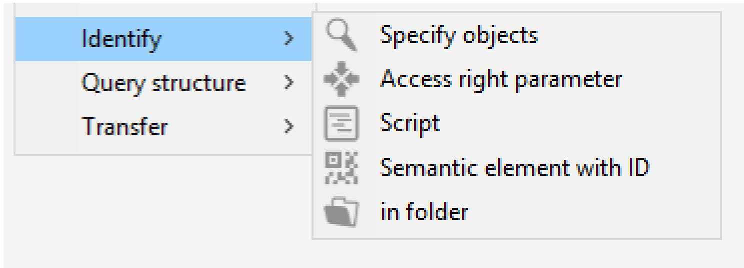
Comparative value results from the script

Attribute value conditions may be removed from partial searches and replaced by a script and attribute condition. The results of the script are then used as a comparative value for the attribute value condition, e.g. if the comparison operators do not suffice for a specific query.

1.3.1.4.3. Identifying objects

The structured query provides several options for identifying objects within the Knowledge Graph. To simplify matters, the previous examples often defined the objects. This type of manual determination may, in practice, be of help in testing structured queries or determining a (replaceable) default for a parameter entry.

At this point we have already become familiar with the combination with the name attribute which can, of course, be any random attribute. In the menu item *Identify* we will find some more options for defining starting points for the structured query:



1.3.1.4.4. Access right parameter

The results of the query may be made dependent on the application context. This particularly applies in connection with the configuration of rights and triggers when, generally speaking, only *user* is usable.

1.3.1.4.5. Script

The objects to be entered at this point are defined by the results of the script.

1.3.1.4.6. Semantic element with ID

You may also determine an object via its internal ID. This condition is normally only used in connection with parameters and the use of the REST interface.

1.3.1.4.7. In folder

Using the search mode *in folder* the contents of a collection of semantic objects can be entered into a structured query as input. The selection symbol will enable you to select a folder within the work folder hierarchy. The objects of a collection are filtered with respect to all other conditions (including conditions for terms).

1.3.1.4.8. Parameter conditions

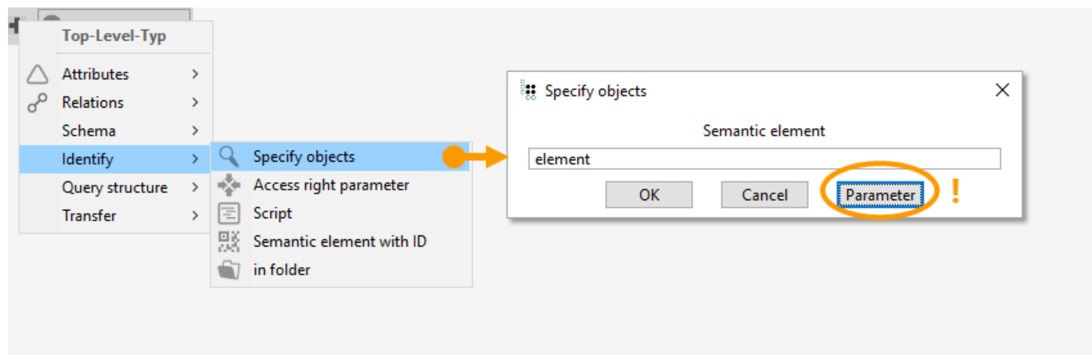
Parameters

By using parameters in a structured query, values can be passed when using the query in JavaScript. Parameters can be freely assigned, a query is query is called as follows:

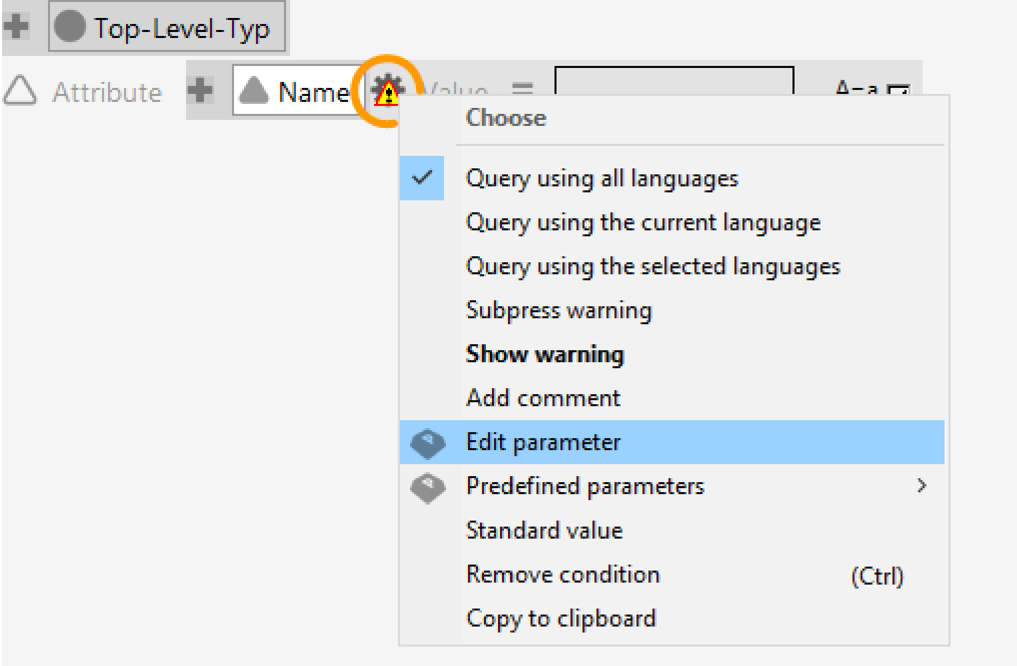
```
$k.Registry.query("<registryKeyOfQuery>").findElements({
  "<parameterNameInQuery>": "<input>"
})
```

The parametrized input can be in forms of:

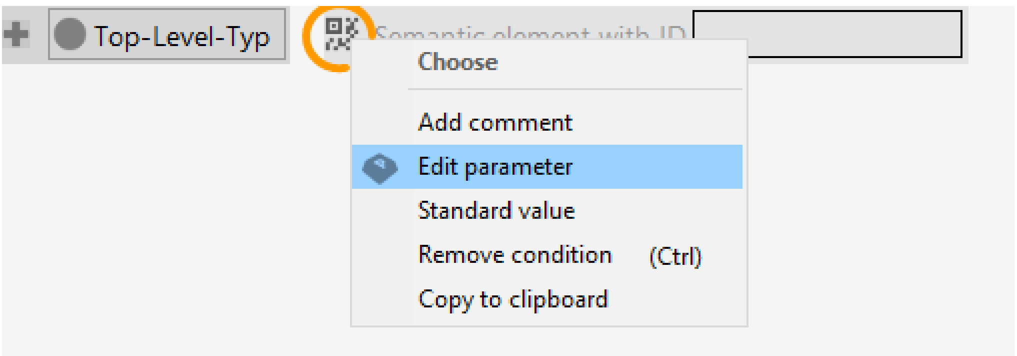
- semantic element



- attribute value



- element id



There are two possibilities to test structured queries using parameters:

- Using the test environment of the structured query
- Invoking the structured query by script (executing or debugging)

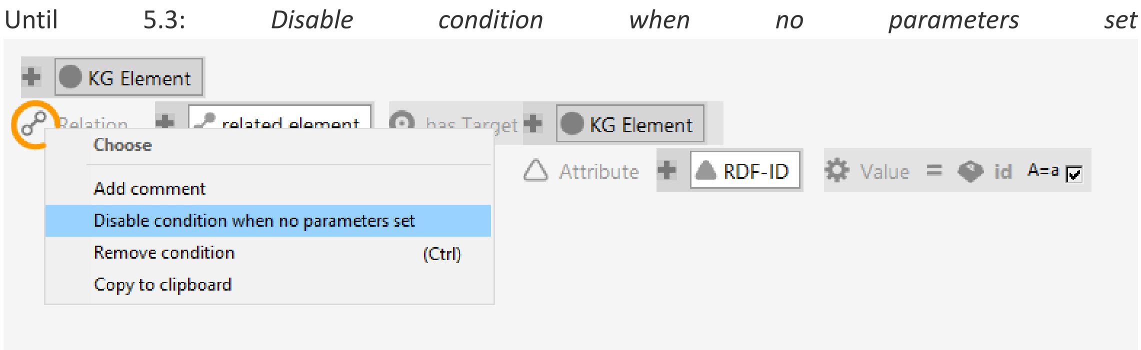
In general, there are four conditions a parameter can have:

- Parameter is set
- Parameter is not set
- Parameter is deactivated
- (Parameter contains empty string)

Optional parameters

The structured query has a feature that allows using optional parameters: for a certain branch of the query, the context menu offers the condition:

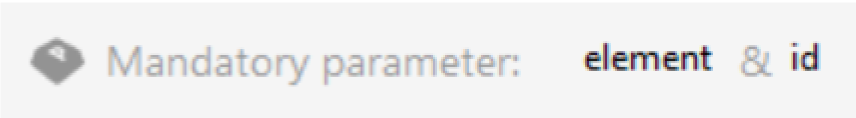
Until 5.3: *Disable condition when no parameters set*



Since 5.4: *Mandatory parameter*

If the optional parameter condition has been set, it has the following effect: From this point on, the rest of the branch (to the right) will not be encountered as condition for the query result when the respective parameter has been deactivated.

If several parameter conditions have been set within one branch, the AND logic applies:



If all mandatory parameters are deactivated, the subsequent query branch will be left out completely when computing the query result, else the parameters which are set are will be used.

NOTE	When the parameter is not set, the test environment will nevertheless throw an error despite the optional parameter condition. If testing of optional parameters is needed, the parameter needs to be **disabled** in order to test an unset parameter

condition.

Important rules about setting parameters

Parameter condition	Setting in structured query	Setting in JavaScript	Result
Parameter is set	Parameter value has been entered	Variable containing parameter value is defined	Parameter condition is encountered in query result
Parameter is **not set**	No parameter value has been entered (just executing query)	Handing over no parameter Using `findElements()` or `findHits()` without arguments or setting parameter to `undefined`.	Error: "Parameter xy is missing"
Parameter is **disabled**	Clicking on x besides parameter	Setting parameter to `null`.	• With conventional parameter: as if the parameter requirement would not exist within the structured query • With optional or mandatory parameter: the branch from the optional condition until end of query branch will be ignored
Parameter contains empty string	Entering `''` or `""`, rejecting search dialog if occurring	Variable for parameter set to empty string `''` or `""`	Query branch will return no result; if no alternatives exist, the whole query might return no results

WARNING

Risk of search results containing false positives

For predictability and reliability of query results in scripts, make sure to avoid parameter values from being `null` inadvertently, since no errors are thrown system wise. Use control structures to catch unattended conditions of parameter input.

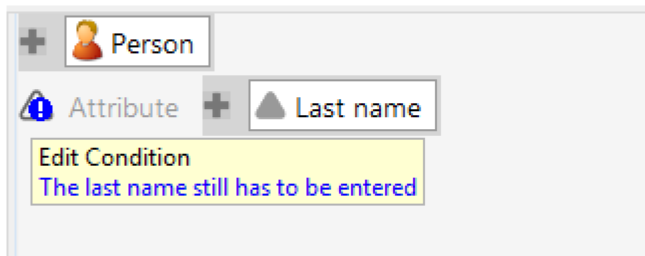
When an optional parameter is passed on to the structured query by means of a script in a *search view* or a *search result view*, the value type of the

parameter also needs to be set to *optional*. If the value type is set to *obligatory*, the structured query will not deliver any search result when the script sets the parameter value to `null` (with the intention to deactivate the optional parameter).

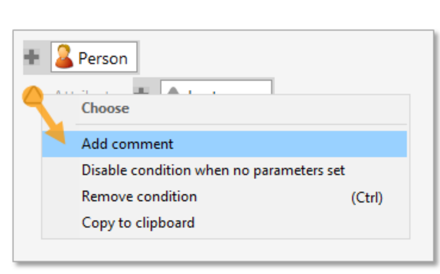
1.3.1.5. Comments in structured queries

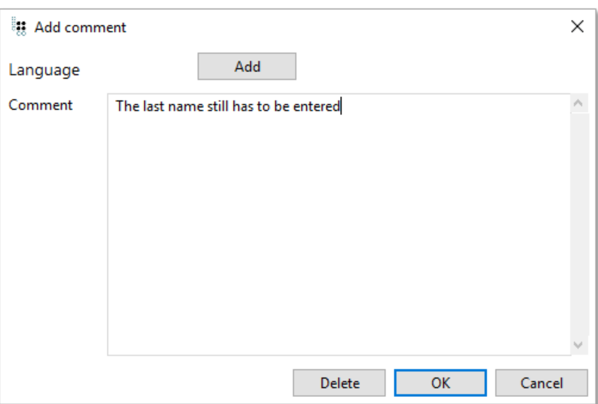
1.3.1.5.1. Adding comments

Every condition in a structured query can be commented. For adding a comment, choose the option *Add comment* in the context menu. At the condition in the structured query, an existing comment causes a blue indicator flag which shows up a text in case of mouseover.

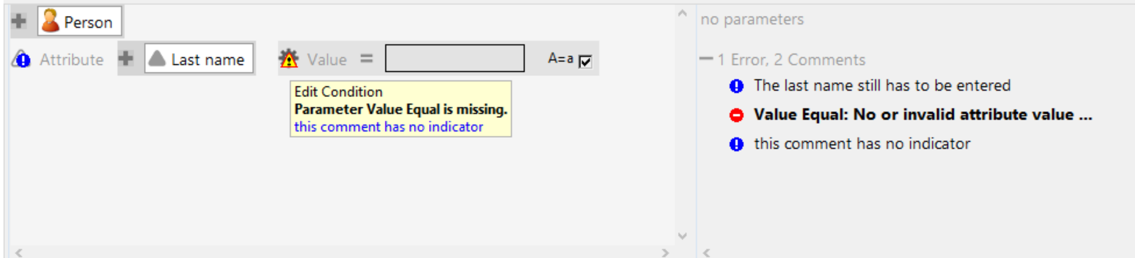


By means of the dialog *Edit comment*, the corresponding comment can be changed or removed:

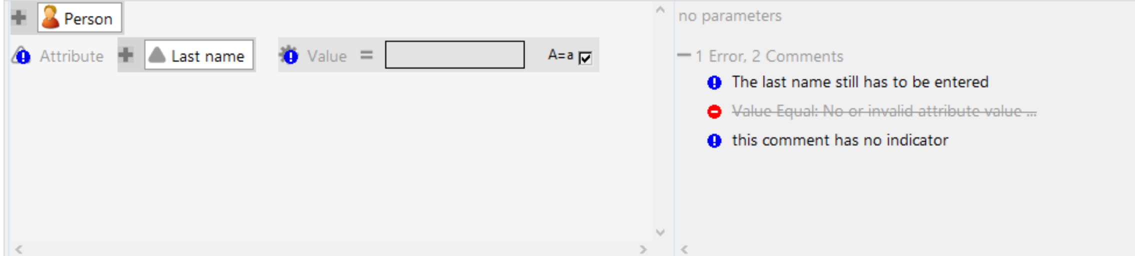




The indicator flag for comments is not shown when the condition has a warning or a fault. In this case you only can see the yellow warning indicator or the red fault indicator. Additionally, all warnings, faults or comments will be listed in their order on the right side below the parameters editor.

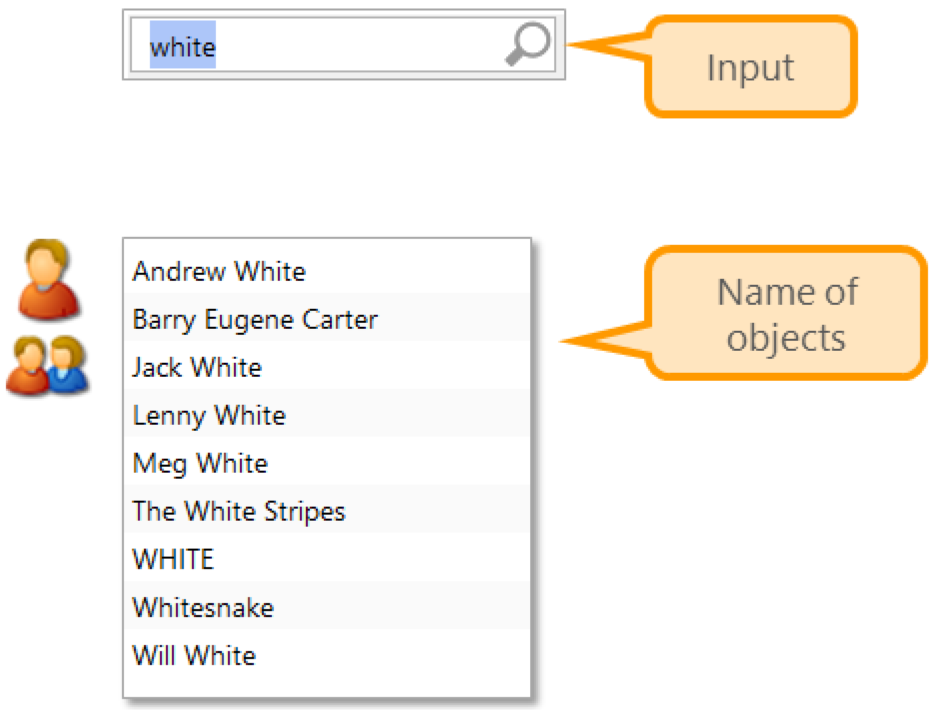


Warnings and cautions can be suppressed in the indicator indication if you want to ignore them at this point (of course, this is not recommended). To do so, click on the indicator symbol in the listed view or choose the function *Suppress warnings* in the context menu of the condition. The indication can be reactivated on the same way or by choosing the context function *Show all warnings* of the root finder.



1.3.2. Simple Search / Fulltext search

Processing the search queries of users may be carried out with or without interaction (e.g. with type-ahead suggestions). The starting point is, in any case, the character string entered. In configuring the simple search we can now define with which objects and in which attributes we search according to the user input and how far we differ from the character string entered. Here is an example:



How do we have to design and organise the search in order to receive the below feedback on objects from the entry "white"? In all cases we will have had to configure the query to show that we only want to have persons and bands as the results. How is it, however, if there are any deviations from the user input?

1. *When is the (completely unknown) Chinese experimental band called "WHITE" a hit?*

   If we state that upper case and lower case doesn't matter

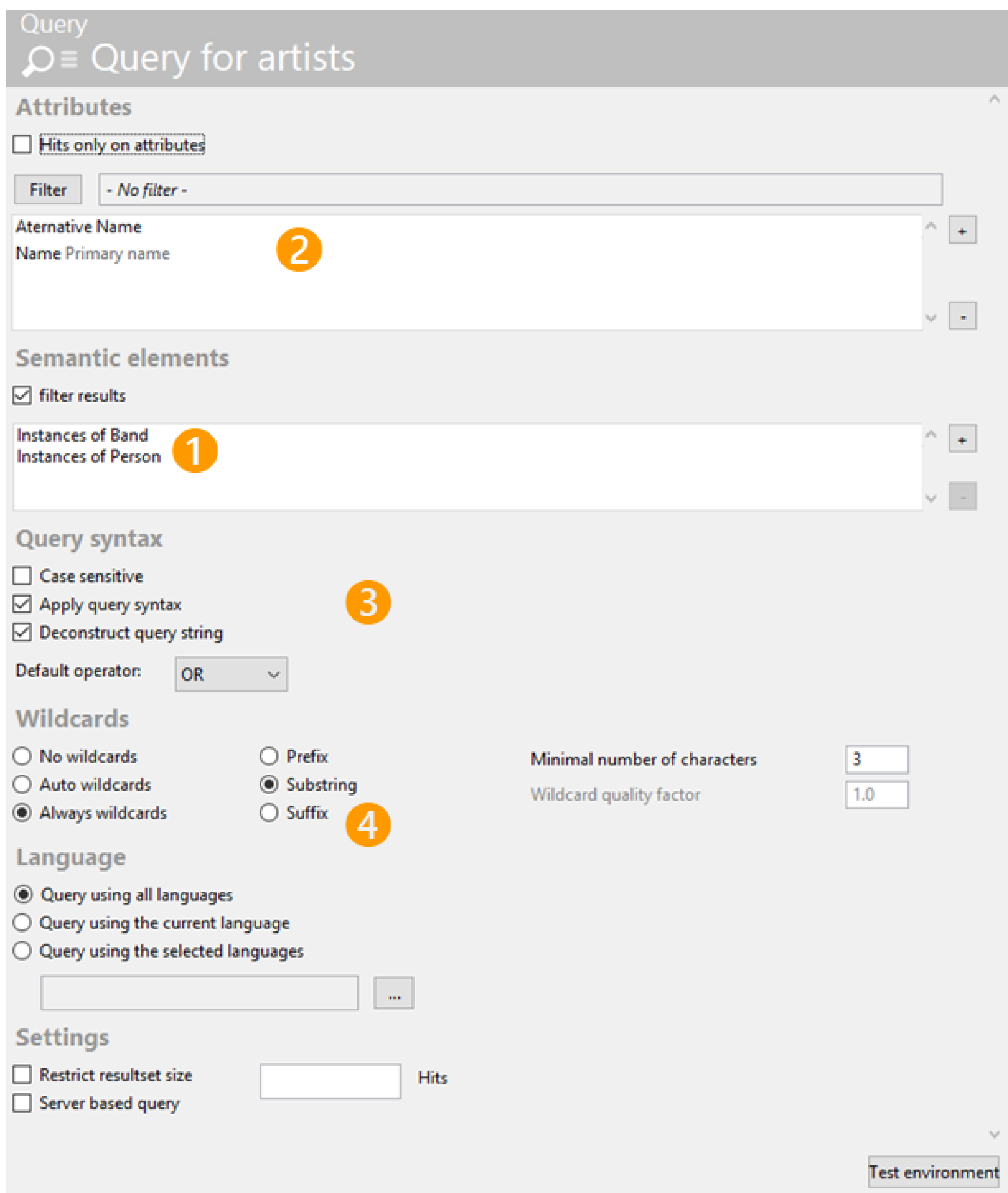
2. *When will we receive "Whitesnake" as a hit?*

   If we understand the entry to be a substring and attach a wildcard

3. *When "Barry Eugene Carter"?*

   If we not only search through the object names but include other attributes as well — his stage name is namely "Barry White".

These options can be found again in the search configuration as follows:



Configuration of the simple search with (1) details as to which types of objects are to be browsed through, (2) in which attributes the search has to be made, (3) upper case and lower case and (4) placeholders.

1.3.2.1. Simple search — details of the options

1.3.2.1.1. Placeholder / wildcard

The entry is often incomplete or we want to retrieve the entry in longer attribute boxes. To do this, we can use placeholders in the simple search. The following settings for placeholders can be found in simple search:

Wildcards

No wildcards | Prefix | Minimal number of characters 3
Auto wildcards | Substring | Wildcard quality factor 1.0
Always wildcards | Suffix

- *Prefix* (trailing placeholder) finds the [White Lies] for the entry "white"
- *Substring* (leading and trailing placeholder) finds [The White Stripes]
- *Suffix* (leading placeholder) finds [Jack White]

WARNING Leading placeholder is slow.

The option *Always wildcards* works as if we had actually attached an asterisk in front and/or behind. Behind *Auto wildcards* there is an escalation strategy: in the case of automatic placeholders, a search is made first with the exact user entry. If this does not deliver any results a search will be made with a placeholders, depending on which placeholders have been set. With the option prefix or substring there is once again a chronological order: in this case you look for the prefix first (by attaching a wildcard) and, if you still can't find anything, you make a search for a substring (by means of a prefix and attaching a wildcard).

If you are allowed to attach placeholders in your search you can state in the box *Minimal number of characters* how many characters the search entry must show to actually add the placeholders. By entering 0 this condition is deactivated. This is particularly important if we set up a type ahead search.

With the *Wildcard quality factor* you can adapt the hit quality to the extent that the use of placeholders will result in a lower quality. In this manner we can, if we want to give the hits a ranking, express the uncertainty contained in the placeholders with a lower ranking.

If the option *No wildcards* is selected the search entry will not be changed. The individual placeholder settings will then not be available.

The user can, of course, use placeholders themselves when entering the search string, which will then be obeyed.

1.3.2.1.2. Apply query syntax

When the box for the option *Apply query syntax* is unchecked, a simplified form of the analysis of the search input is used in which, for example, the tokens | (OR condition), & (AND condition), and ! (negation) no longer have an effect on the result. Nevertheless, in order to be able to define how the hits for the tokens should be compiled, the *Default operator* can be switched to *AND* or *OR*. What applies to all linking operators is the fact that they do not refer to values of individual attributes, but to the result objects (depending on whether *Hits only on attributes* has been set). A hit for "online system" thus delivers semantic objects which have a matching attribute for both "online" and "system" (which is not necessarily the same).

1.3.2.1.3. Filtering

Simple searches, full-text searches and also some of the specialized searches may be filtered according to the types of objects. In the introductory example, we made sure that the search results only included persons and bands. Attributes which do not match a possible filtering are depicted in red bold print within the search configuration dialogue. In our case this could be an attribute "review", for example, which is only defined for albums.

1.3.2.1.4. Translated attributes

In case of translated attributes we can neither select a translation, nor have the language dynamically defined. Search for multilingual attributes, then in the active language or in all languages, depending on whether the option *Query using all languages* is selected.

1.3.2.1.5. Query output

A maximum query output may be defined by entering the maximum number in the *Hits* box. By toggling the checkbox `Restrict result set size` the mechanism can be activated or deactivated.

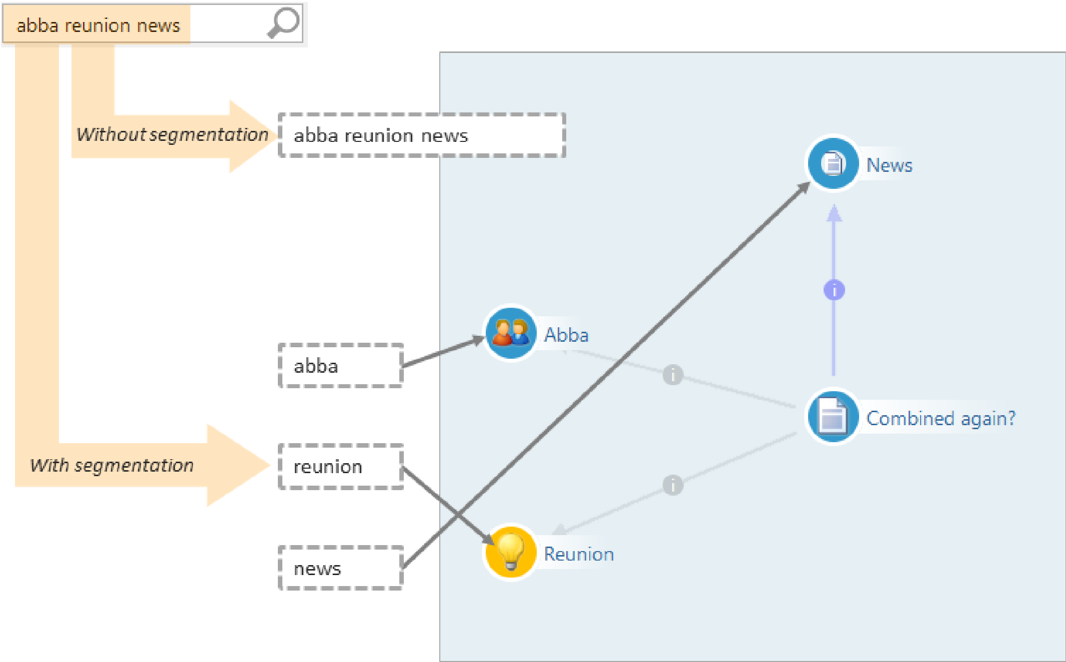
WARNING	If the number is exceeded, no output will be shown!

1.3.2.1.6. Server-based search

Generally speaking, each search can also be carried out as a server-based search. The prerequisite for this is that an associated Job-Client is running. This option can be used when it can foreseen that very many users will make search queries. By outsourcing certain searches to external servers, the i-views server will be disburdened.

1.3.2.2. Multi word search inputs

In our examples for queries the users have, until now, only entered one search term. However, what would happen if the user entered "Abba Reunion News", for example, and thus would like to find a news article which is categorized by the keywords "Abba" and "reunion"? We have to disassemble this entry because none of our objects would match the entire string or at least not the article being searched for:



Our examples so far do not, however, fall short only due to multi word search inputs. We also often have search situations in which it does not make sense to regard the names or other character strings from the Knowledge Graph, with which we compare the input, as blocks, e.g. because we would like to retrieve input in a longer text. In this case the wildcards will eventually no longer be an adequate means: if we also want to disassemble the input on the page of the object and the text attributes which have been searched through it would be better to use the full-text search.

1.3.2.3. Full text search and indexation

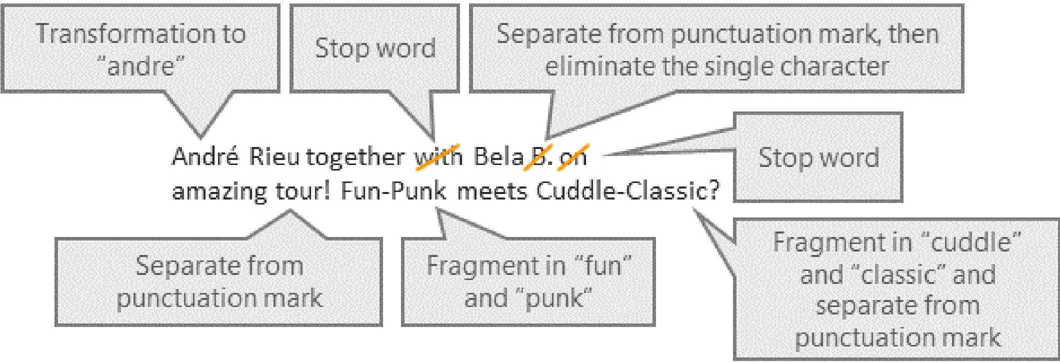
If we want to view or search through longer texts word by word, e.g. description attributes we recommended the use of full-text index. What does something like that look like?

Term	Occurrence
aaliyah	Doc#155, Pos. 548644 / Doc#459, Pos. 934875 / Doc#935, Pos. 26526
abba	Doc#132, Pos. 43095 / Doc#459, Pos. 46795 / Doc#935, Pos. 534955 / Doc#353, Pos. 367773 / Doc#711, Pos. 92634
abbey	Doc#464, Pos. 95367 / Doc#2543, Pos. 65258 / Doc#634, Pos. 35241
abbreviation	Doc#436, Pos. 54362
abbreviator	Doc#463, Pos. 234652
abnormity	Doc#253, Pos. 4652
abo	Doc#234, Pos. 32243 / Doc#332, Pos. 23414

The full-text index records all terms/words which occur within a portfolio of texts so that i-views can quickly and easily look up where a particular word can be found in which texts (and in which part of the text).

"Texts", however, are not usually separate documents within i-views, but the character string attributes which have to be searched through. Their full-text indexing is a prerequisite for the fact that these attributes are offered in the search configuration.

Even full-text indexing concerns the deviations between the exact sequence of characters within the text and the text which is entered in the index and which can hence be retrieved accordingly. An example of this: a message from the German music scene:



In this example we find a small part of the filter and word demarcation operations which are typically used for setting up a full-text index:

1.3.2.3.1. Word demarcation / tokenizing

Often in punctuation such as exclamation marks are placed directly on the last word of the sentence without a space in between. In the full-text index, however, we want to include the entry {tour}, not {tour!} — hardly anyone will search for the latter. For this purpose, when setting up the full-text index we have to be able to specify that certain characters do not belong to the word. The decision is not always so easy: In a character string such as "hard-rock" which occurs in a text we have to decide whether we want to include it as an entry in the full-text index or as {hard} and {rock}. In the first instance our message will then only be found if an exact search is made for "hard-rock" or, for example, "*ard-ro*", in the second instance for all "rock" searches.

What we will probably keep together in spite of the occurrence of punctuation, i.e. exclude from tokenizing, are abbreviations: when AC/DC come to Germany o.i.t. (only in transit) it is probably better to have the abbreviation in the index instead of the individual letters.

1.3.2.3.2. Filter

By using filter operations we can both modify words when they are included in the full-text index and also completely suppress their inclusion. We can maintain a list of *stop words*. Moreover, we probably do not want individual characters (Bela B.) to be in the index like this — the likelihood of confusion is too great. Using other filters we can restore words to their basic forms or define replacement lists for individual characters (e.g. in order to eliminate accents). Other filters, in turn, clear the text of XML tags.

We can set all this in the Knowledge Builder within the global settings via *Index configuration* > *Indices*. We can then assign these configurations to the character string attributes. The index configuration is organized in such a manner that filtering can take place before the word demarcation and after the word demarcation.

The full-text search does not affect the wildcard automatism of the other queries but the user may, of course, provide his input with wildcards.

1.3.3. Search pipeline

Search pipelines enable individual components to be combined to complex queries. Single components perform operations in the process, e.g.:

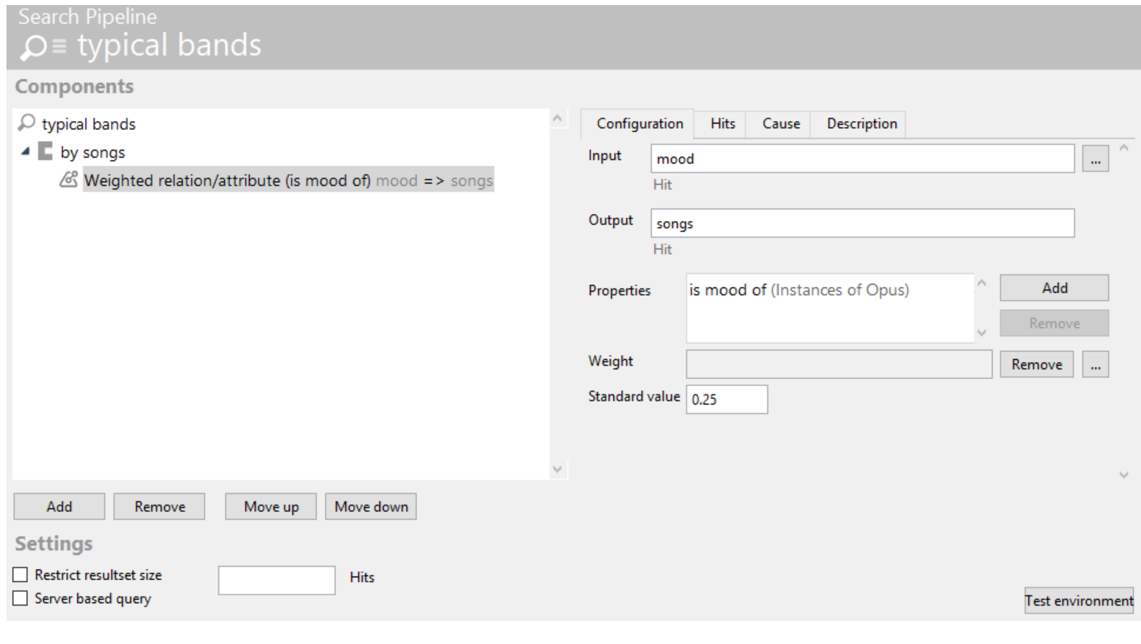
- traversing the Knowledge Graph and thus determining the weighting
- performing structured queries and simple queries
- compiling hit lists

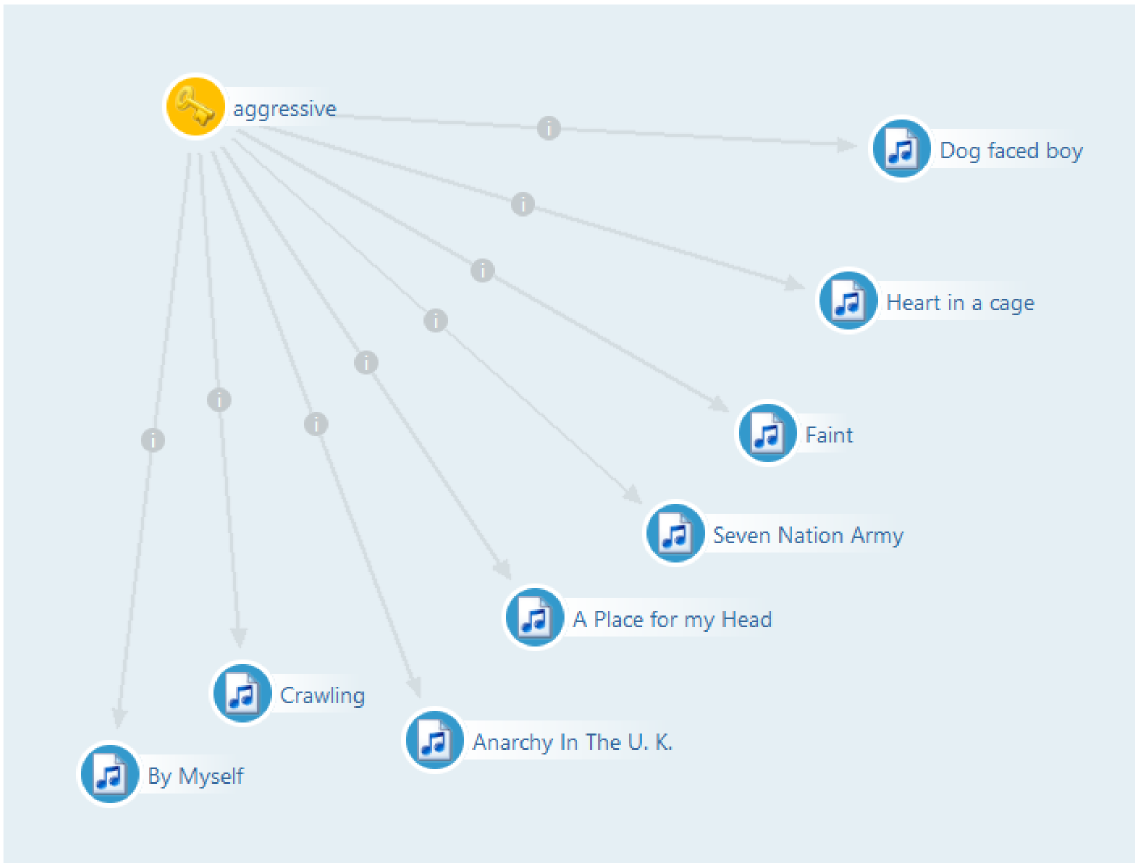
Every query step produces a query output (usually a number of objects). This query output may, in turn, be used as input for the following components in the pipeline.

1.3.3.1. Example

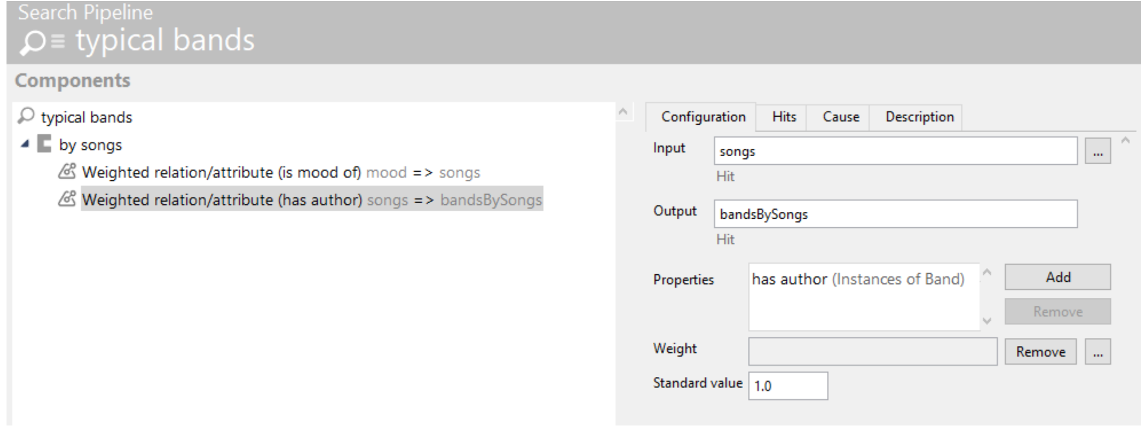
Let us assume that songs and artists from our music graph are characterized with tags named moods. Based on a certain mood we now want to find which bands best represent this mood.

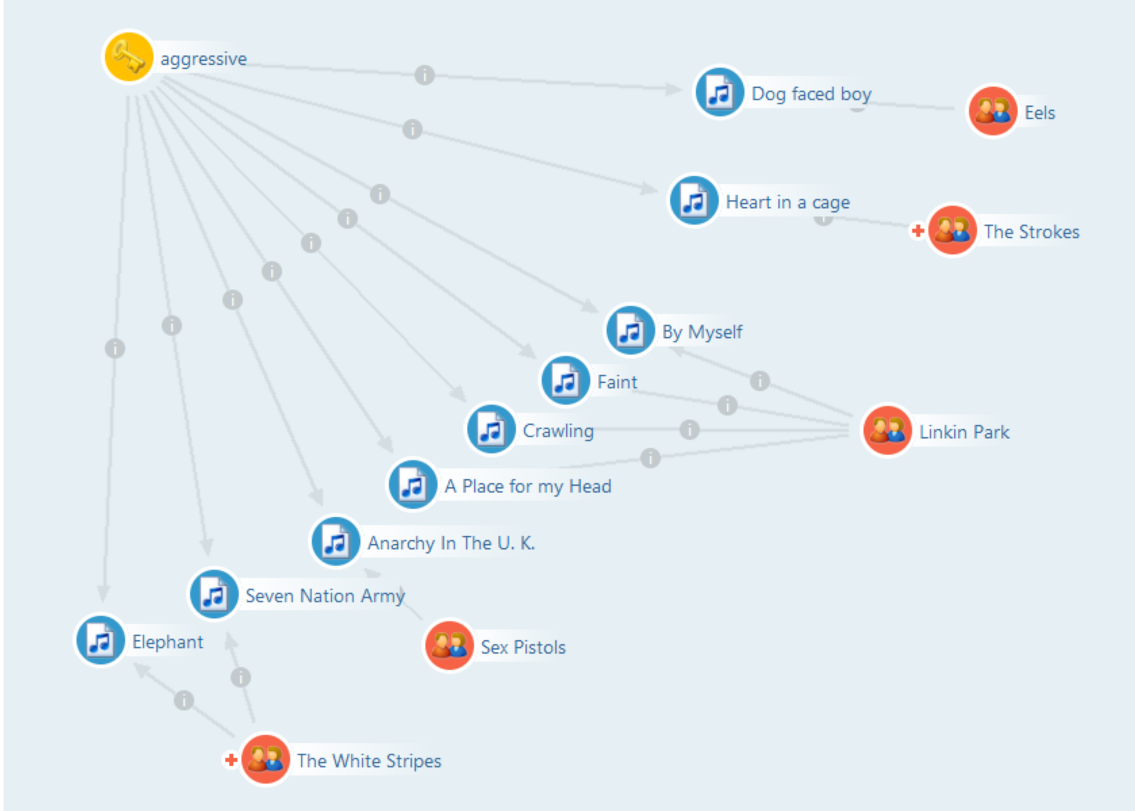
Step 0 of our search pipeline accepts a mood as parameter and assigns it to the variable named "mood". In this example, we use the mood "aggressive" as input and see how the pipeline can be used to find bands typically associated with this mood. In step 1 the pipeline goes from the starting mood to the songs which are assigned to the mood "aggressive" via the relation *is mood of*:



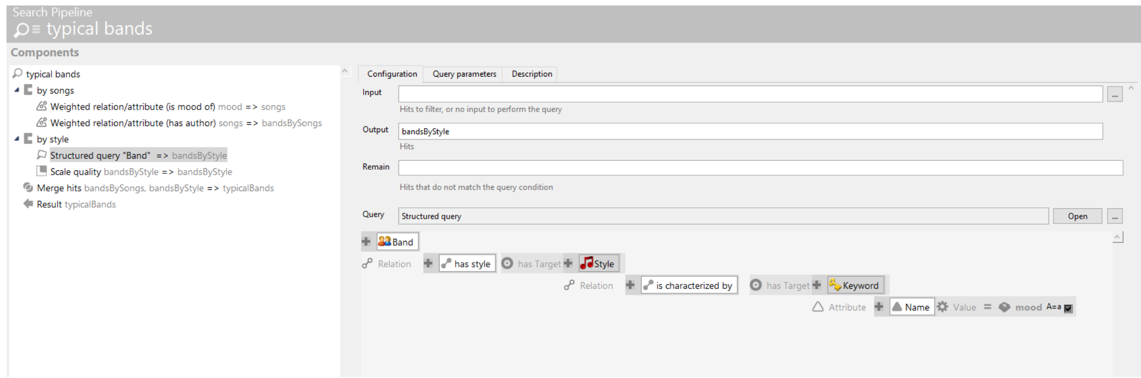


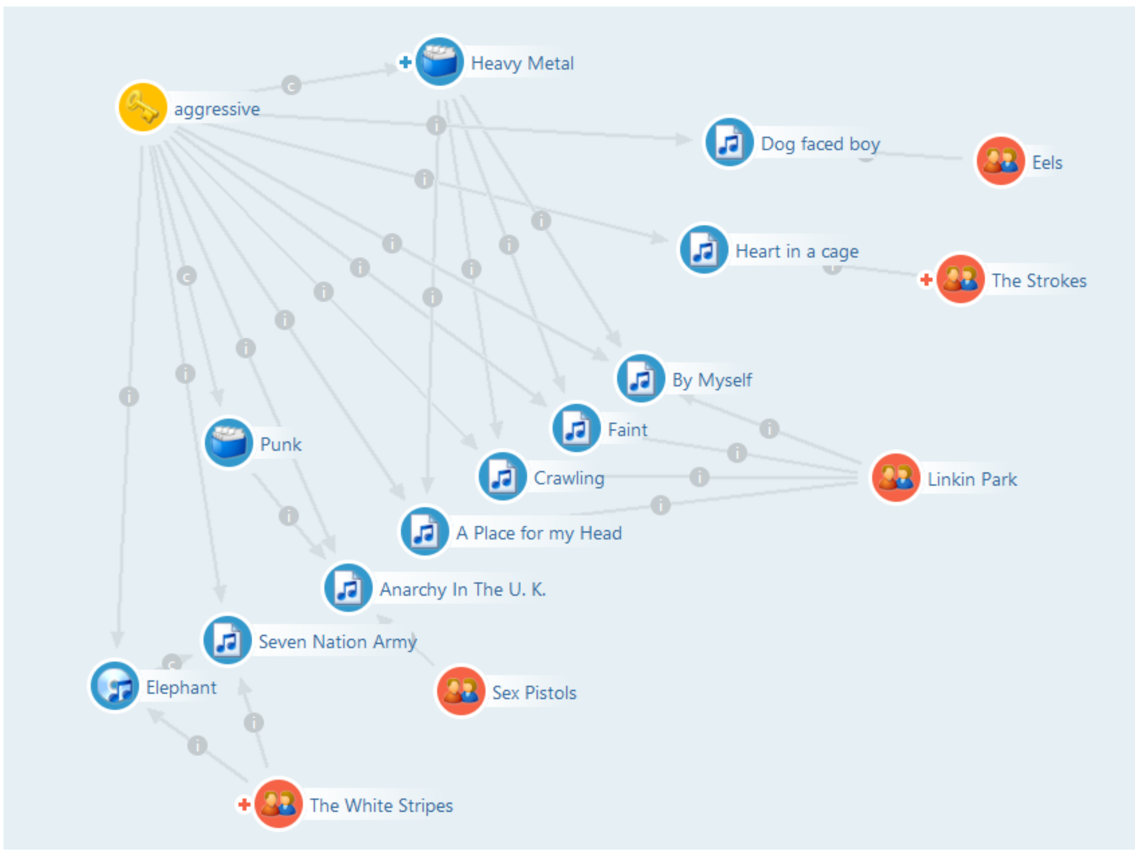
In the second step we go from the number of songs detected in the "mood" searched for to the corresponding bands via the relation *has author*:



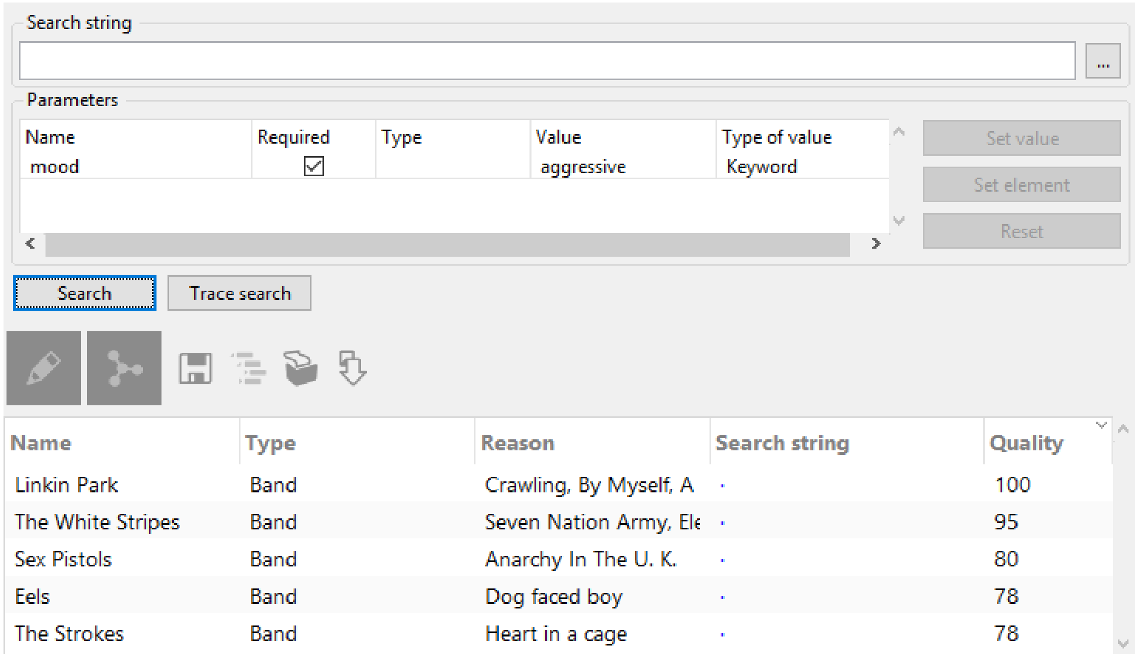


Now we would like to pursue a second path: from the starting point "mood" "aggressive" to the musical directions which are characterized by aggressiveness. Based on this number of relevant musical directions we have to go to bands which are assigned to this mood. We go down this alternative path in one step using a structured query:





From the last two steps we give the indicator "musical direction" a somewhat lower weighting and compile the outputs at the end:



The steps are processed in sequence: the input and output define which step will continue to work with which hit list. For instance, in this manner we would be able to begin again with "mood" on

our alternative path.

1.3.3.2. The principle of weightings

It was the goal to give the bands we obtained as outputs a ranking which shows how great their semantic "proximity" is to the mood aggressive. In particular, we influence ranking in this search at two positions: right at the end we weight bands higher in the summary which are found both via their musical direction and their songs. In this case this applies to Linkin Park and the Sex Pistols. The higher ranking of Linkin Park results from the fact that again and again different songs lead to Linkin Park with the mood aggressive. Since more aggressive songs from Linkin Park are in the database, Linkin Park should be "rewarded" with a higher ranking.

1.3.3.3. Configuration of search pipelines

The individual components of a search pipeline are depicted in the main window in the box *components* in the order of sequence in which they are implemented.

Using the button *add* we can insert a new component at the end of the existing components.

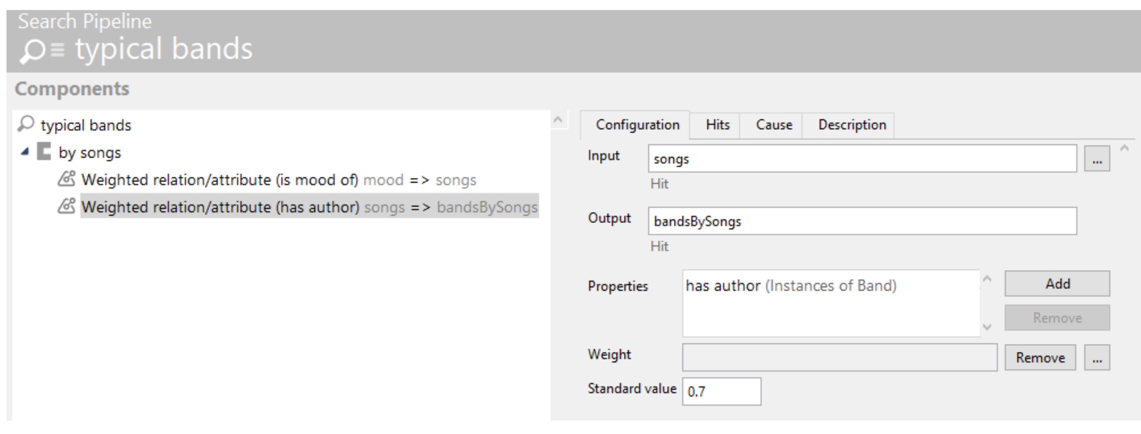
Grouping with blocks serves only to provide an overview, e.g. for the compilation of several components in a functional area of the search pipeline.

The order of sequence of the steps can be changed using the button *upwards* and *downwards* or with drag & drop.

Using the button *remove* the component selected will be removed, to include all possible sub components. The configuration for the component selected is displayed on the right-hand side of the main window.

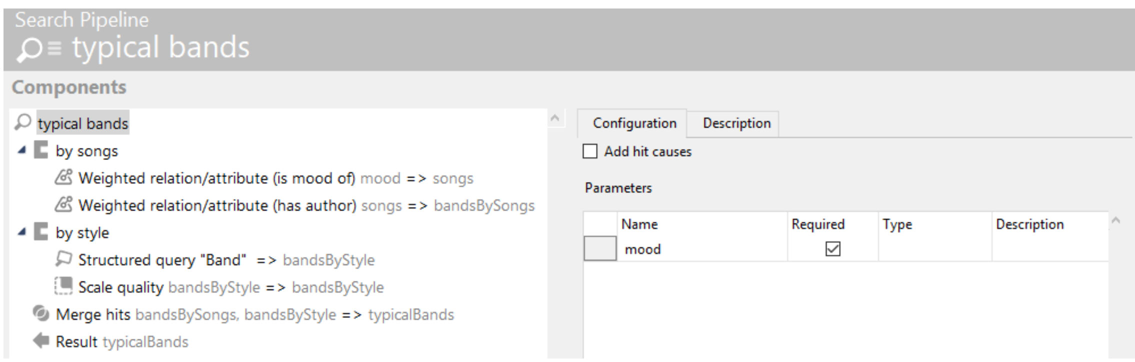
1.3.3.3.1. Configuration of a component

A selected component may be configured on the right-hand side of the main window using the tab *Configuration*: most components need input. This usually comes from a previous step. In this way, the first components in our example pass on the output under the variable "songs" to the next component, this then goes from there to the bands and, in turn, gives the output to the next steps as "bandsThroughSongs":



Using the input and output variable we can also, in later steps, re-set to the initial output which we saw in the last paragraph.

We define the input parameters as global settings for the search. Under the name which we assign here we can then access these inputs in our search pipeline during each step. In our example the input parameter for identifying typical bands is the mood.



Some components enable a deviation from the standard processing sequence:

Individual processing

Elements of a set, e.g. hits from a search may be processed individually. This is practical if you want to assemble an individual environment of adjacent objects for search hits. In individual processing each element of the configured variable in the single hit is saved and implemented in the sub components.

Condition for set parameters

This component only carries out further sub components if predefined parameters have been set, whereby the value is insignificant. New sub components may be added by using the "add" tab.

KPath condition

By using a KPath condition we can determine that the sub components may only then be implemented if a condition expressed in KPath is fulfilled. If the condition is not fulfilled the input will be adopted. KPath is described in the *KPath* chapter of the technical manual.

Output

We can stop the search at any stage and return the input. This component is also useful for testing the search pipeline.

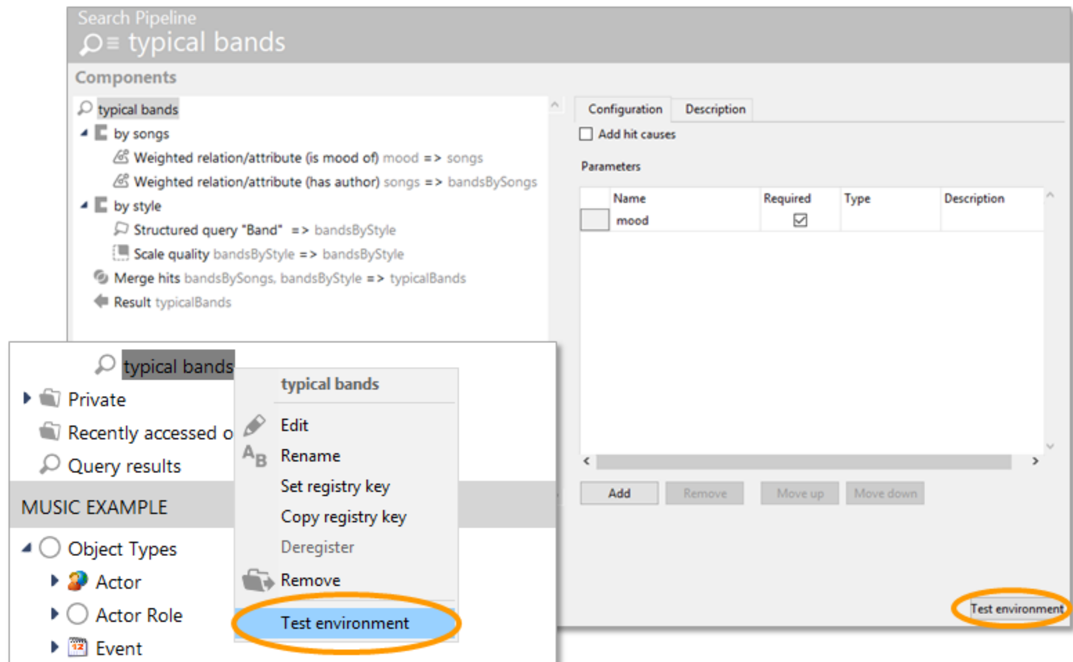
Block component

The block components which we have also used in our example group a lot of individual steps. In order to maintain an overview in extensive configurations we can also change the name of the component using the tab *Description* and add a comment as well. Neither the block components

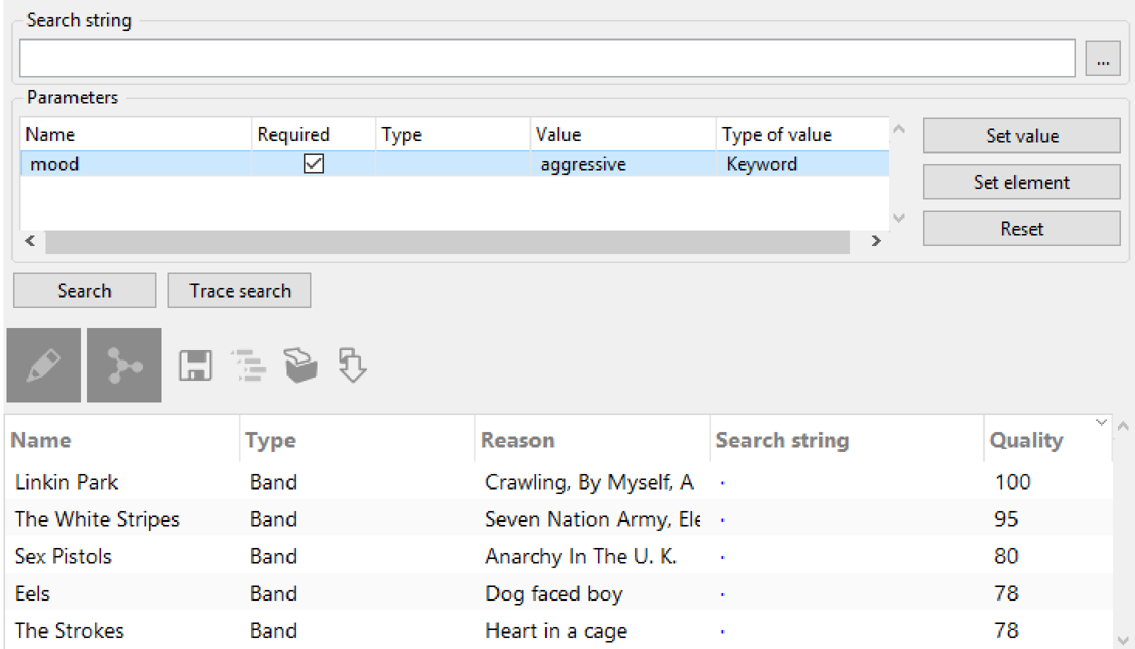
nor the description have any functional effects. Both of them only serve the "legibility" of the search pipeline.

1.3.3.3.2. Test environment

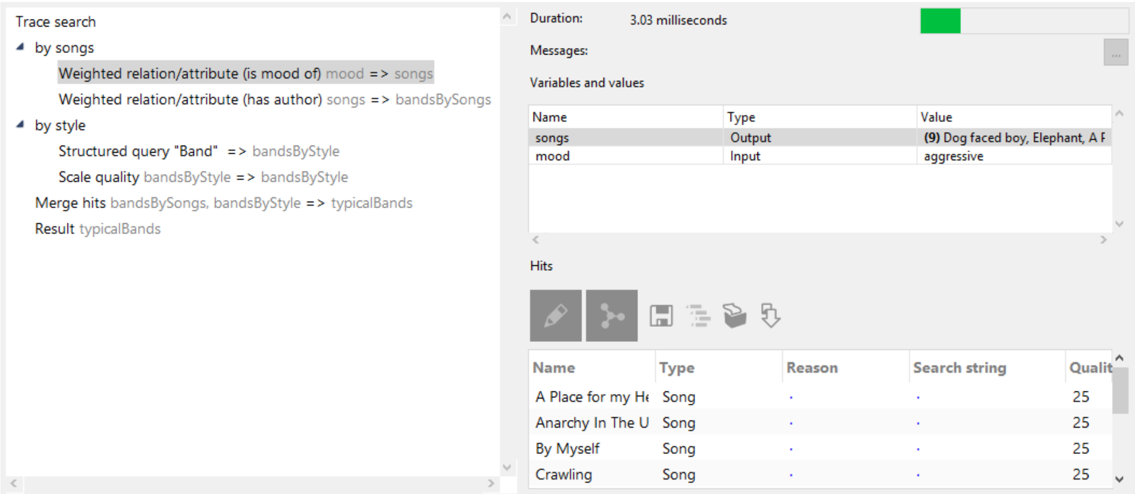
The test environment can be invoked by several ways:



Using the test environment in the menu we can analyze the functioning of the search. The upper section contains the search input and the lower section the output. The input may be a search text or an element from the Knowledge Graph, depending on which required and optional input parameters we have globally defined in the search pipeline. If we wish to enter an element from the Knowledge Graph as a starting point we select the corresponding parameter line and add an attribute value or a (Knowledge Graph) element, depending on the type.



On the tab *Trace search* a report of the search will be displayed. This primarily consists of the configuration of the output variables and the duration of the implementation of each component. The log begins with the pre-configured variables (search string) as well as active users.



1.3.3.3.3. Calculation possibilities

In the case of some components it is possible to summarize several quality values into one single quality value — e.g. in *summarize hits* but also when traversing the relations (see example above). For this purpose the following methods of calculation are available:

- Addition / multiplication
- Arithmetic average / median
- Minimum / maximum

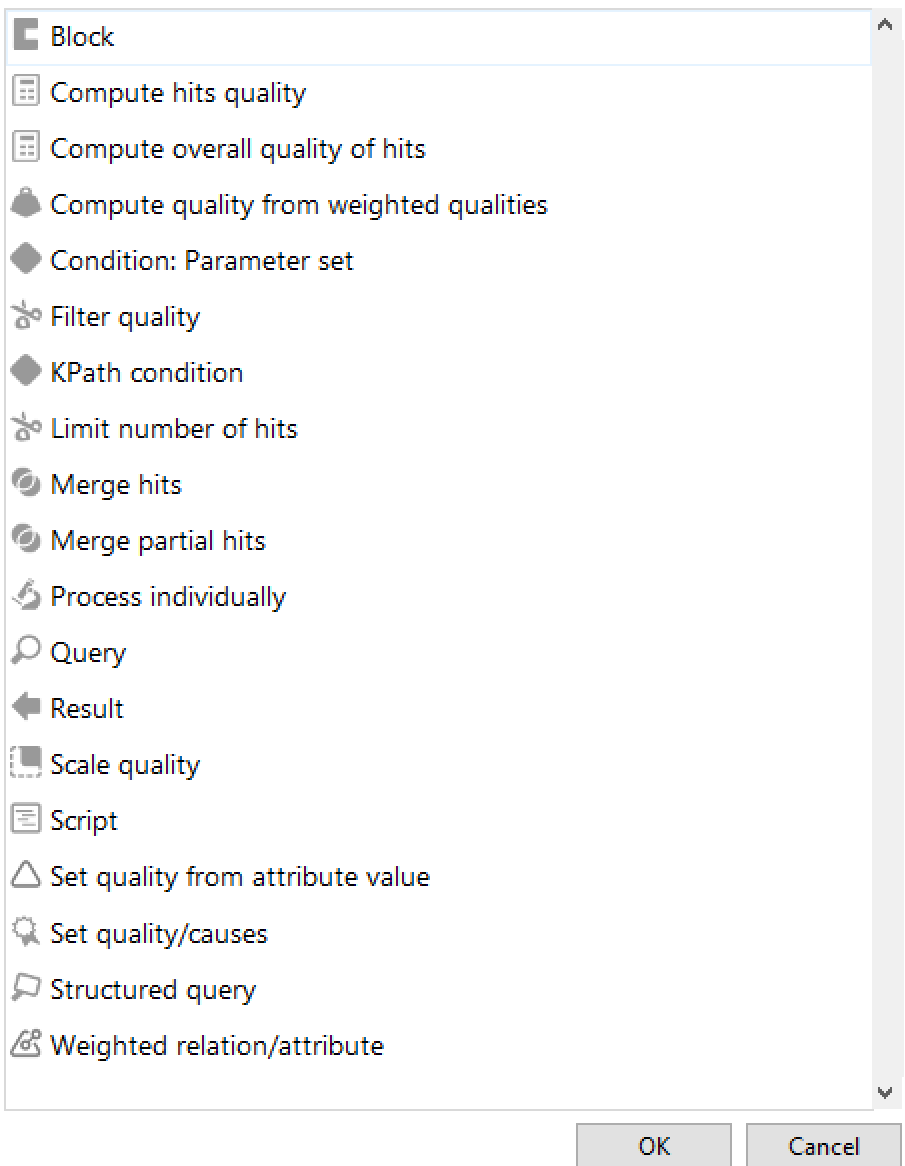
- Ranking

The option *Ranking* is suitable when we want to assemble an overall picture from individual references, e.g. if we want to calculate many paths, at least partially independent paths — at the end still with differing lengths — to an "overall proximity". Using the ranking calculation we ensure that all positive references (all independent paths) keep increasing their similarity without exceeding 100%.

In the search pipeline quality values are always specified as floating point numbers. The value 1 thereby corresponds to a quality of 100%.

1.3.3.4. The individual components

All elements that can be added to the search pipeline either incorporate a structural function, a query function, a logical function or functions for computing qualities:



1.3.3.4.1. Block

The block element is only for optical reasons. To keep larger search pipelines clearly arranged, it can be used to structure several succeeding elements into logical groups. To do so, simply drag&drop the elements underneath the block. The block has no influence on the results of the search pipeline.

1.3.3.4.2. Weighted relation/attribute

Starting with semantic objects, we can traverse the graph in this step and collect relation targets or attributes. To do so, we have to specify the type of relation or attribute.

NOTE	Only collected targets are output, rather than the initial set. If this is to be displayed, we then have to enable the option *Add source hits to result* at the *Hits*

tab.

When traversing a relation, the weighting of hits can be influenced. Let’s assume we want to semantically enhance the "initial mood" of our example search with "sub-moods". But this indirection is to be reflected in a ranking: Connections to bands that run via sub-moods are not supposed to count as much as connections via an initial mood. For this purpose, we can assign a fixed value — e.g. 0.5 — for moving along the relation and then merge input quality, e.g. multiply it. In this case the sub-moods added in this step count only half as much as direct moods.

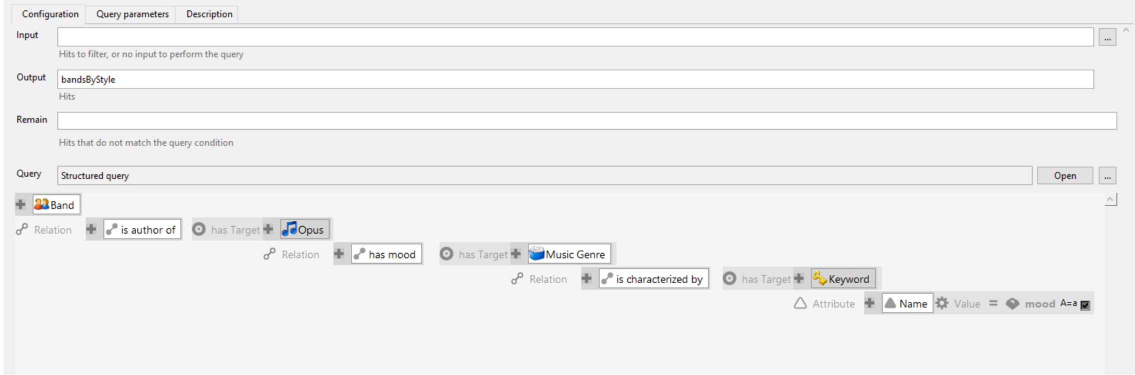
Instead of assigning a fixed weight for moving along the relation, we could also read the value from a meta-property of the basic type float of the selected relation. If the attribute is not available or no attribute has been configured, the default value is used. The value should be between 0 and 1. The hit generation can be configured in detail: For relations, you have the option to also generate a new hit for the source of the relation (rather than for the relation target).

If a relation has been selected as a property and hits are generated for relation targets, we can also transitively trace the relation. The quality value is reduced with each step until the value falls below the specified threshold. If an object has more relations than specified under maximum fan-out, these relations are not traced. The higher the damping factor, the more the quality value is reduced.

1.3.3.4.3. Structured query

We can use structured query components to either search for semantic objects/go from an existing set to other objects (as with the weighted relation) or filter a set.

If we search for objects, we forward our initial set of hits from a preceding step into the search via the parameter name. (In general: Within the expert query, variables of the search pipeline, e.g. search string, can be referenced via parameters.) In this case, the input stays blank.



For filtering, in contrast, we specify a set of objects as the input. The output contains all objects that meet the search condition. Objects that do not meet the search condition can optionally be stored in an additional variable (Rest).

We can either define the structured query ad hoc directly in the component or we can use an existing structured query.

NOTE If an existing search is selected, no copy is created. Any changes to the structured query that we make for search pipeline purposes also modify the query for all other uses.

1.3.3.4.4. Query

You can use the *Query* component to execute simple queries, full text queries and other search pipelines. Simple queries and full text queries receive a string here, e.g. the *search string*: This is a parameter that is available for processing user input in all search pipelines. The hit list of the called search fills the output of this component.

By integrating search pipelines into other search pipelines, we can factorize sub-steps that occur more frequently. Several parameters and entire sets of hits can be transferred to other search pipelines. With integrated search pipelines we can also replace several parameters, that is, we can access of every sub-step output in the integrated search and vice versa. If we go to *Selected parameters*, we can also rename them, for example, if we want to use a set of hits from the integrated search but have already used the name. Alternatively, we can also apply only some of the parameters from the integrated search in order to avoid such conflicts.

1.3.3.4.5. Merge hits

We can use this component to summarize different sets of hits from previous steps. The following methods are available for summarizing:

Join

All hits that occur in at least one of the sets are output as a result

Intersect

Only hits that occur in all sets are output as the result.

With joins and intersects, a semantic object can occur in several sets of hits and has to be computed as one total hit with a new hit quality. The aforementioned calculation options are also available here.

Difference

One of the sets of hits must also be defined as the initial set. The other sets are deducted from this set.

Symmetric difference

The result set consists of objects that are included in exactly one subset (= everything except for the intersection, when there are two sets).

Three different types of total hits can be generated. The selection is particularly relevant if partial hits include additional information.

- To generate uniform hits, remember the original hits as the cause: New hits are generated that contain the original hit as the cause.

- Extend original hits: The original hit is copied and receives a new quality value. If there are several hits for the same semantic object, a random hit is selected.
- Generate uniform hits: A new hit is generated. The properties of the original hit are lost.

1.3.3.4.6. Condition: Parameter set

This element assures that its subcomponents are only processed if preset query parameters (= input for the query elements) are set. The respective parameter can be assigned within the configuration tab. If the parameter is not set, the affected part is skipped and the next part will be processed.

1.3.3.4.7. Process individually

This element is used for processing several hits (= array of hits) in order to use each single hit (= nth element of the hit array) as an input for query elements that only can process a single hit at once. This comes into account for queries expecting a string as input.

> The hits of a preceeding query are intended to be processed by a simple query. To do so, we process these hits individually: The hits, which are passed on in an array, will be split up again into individual array elements (for more information, see chapter *Model "hit"*). Due to the fact the single hit itself is consisting of a semantic element, its hit quality, its hit cause and possibly further user-defined query properties, a script is needed to return the name of the semantic element of the hit. The returned name string then can be used as an input for a simple query. The hits of the simple query for each input element can be merged again into a single hit array by the query element *Merge partial hits*.

NOTE To merge the hits from the element *Process individually* an to calculate the overall hit quality correctly, the element *Merge partial hits* is needed.

1.3.3.4.8. Merge partial hits

During individual processing you frequently have to generate a total set from partial hits. The component *Merge partial hits* enables you to do so. This summarizes all hits of one or more partial sets of hits ("hit collections"). The difference to *Summarize hits* is that summarizing only takes part at the end, not for every partial hit set. This is relevant in particular when calculating the quality because summarizing hits would return incorrect values, in particular for the computing method *Median*.

1.3.3.4.9. Script

A search pipeline can contain a JavaScript. This can access the variables of the search pipeline. Furthermore, a script can transfer several parameters to the search pipeline. The result of the script is used as the result of the component.

JavaScript API is described in a separate manual.

1.3.3.4.10. Set quality/causes

For hits arising as a result caused by the input from preceding (and also distant) query elements, dedicated quality values or indirect causes might be needed which otherwise might be missing.

Setting individual qualities comes into account especially when a structured query has been used before: Structured queries always return hits with the hit quality 1.0 (100%) due to the fact that the hits arise wether a structural relationship could be found or not — in this case, existence is no gradual match (like the output of a string-processing query). By setting the output parameter from query elements positioned before the structured query as a source of quality information, an "interconnection" can be built to recapture individual quality values again. If just the overall quality needs to be adjusted, the query element *Scale quality* is adequate here fore. If a quality influence per relation distance is needed, the query element *Weighted relation/attribute* is more suitable.

Setting the causes of hits comes into account when not the direct causes, but distant causes from another (preceding) query are needed. By setting causes, the hits can be "explained" in forms of a graph: The resulting semantic elements, their originally causing elements and all intermediate semantic elements will be shown at once.

1.3.3.4.11. Set quality from attribute value

For hits, we can copy the quality value from an attribute of the semantic object. If the object does not have exactly such an attribute, the default value is used. The value should be between 0 and 1.

1.3.3.4.12. Compute quality from weighted qualities

To adapt the quality of a search hit, it can be useful to compute a total value from individual partial qualities. The qualities must be available as numeric values. These values are used to calculate a new total quality.

1.3.3.4.13. Compute overall quality of hits

You can use the individual quality values of a set of hits to compute a total quality.

1.3.3.4.14. Filter quality

We can restrict sets of hits to hits whose quality value falls within specified limits (minimum or maximum). Normally, we want to filter out hits that fall below a certain quality threshold.

1.3.3.4.15. Limit number of hits

If the total number of a set of hits is to be restricted, we can add the component *Limit number of hits*. We can use the option *Do not split hits of the same quality* to prevent a random selection in case of several hits of the same quality in order to comply with the total number. We then get more hits than specified.

In some very specific cases, we can also randomly select the hits, e.g. if we have a large number of hits with the same quality and want to generate a preview.

1.3.3.4.16. Scale quality

Die quality values of a set of hits can be scaled. A new set of hits with scaled quality values is calculated. The calculation takes place in two steps:

1. Die quality value of the hits are limited. The threshold values can either be specified or calculated. The calculation determines the minimum and maximum value of the hits. If thresholds are specified and a hit has a quality value that falls outside of the thresholds, the value is limited to the threshold value. If you want to remove such hits, you have to execute the restrict quality component first. Example: Mapping percentage values to school grades. 30% is average, over 90% is high score. The values can be scaled linearly from 30% to 90%.
2. Following that, the quality values are scaled linearly. Hits with the minimum/maximum input value receive the minimum/maximum scaled value.

1.3.3.4.17. Compute hits quality

You can use a KPath expression to generate a new hit with calculated quality for a hit. The KPath expression is calculated on the basis of the input.

1.3.3.4.18. Result

The *Result* element is used to determine at which position of the search pipeline processing ends and which parameter value is to be returned as result. Everything underneath the result element will not be processed.

This comes in handy when elements of the search pipeline are momentarily not needed: They simply can be "parked" underneath the result element.

1.3.3.5. KPath

KPath allows addressing of objects within the Knowledge Graph. The notation is similar to XPath but differs in some respects.

The individual elements of the expression normally are separated by a slash `/`. If a KPath expression begins with `/`, then the evaluation starts at the root type, else it starts at the current object (depends on the context of the evaluation). If an element does not correspond to one of the listed elements of the table, it will be interpreted as a name of a sub type. Simple names can be specified without quotation marks.

When specifying a language, it must be stated according its ISO 639-2 code ("eng" for English, "ger" for German, ...).

`@Name`

Attribute "Name"

`//book\Faust/~author`

Relation "author" of the book "Faust"

`//$artifact$/book{eng}`

Sub type "book" (English name) of the type "artifact" (internal name)

`//book*[~author/target()/@Name = "Goethe"]`

All books which had been written by Goethe

1.3.3.5.1. Names

In combination with `@`, `/`, `//``, `\` and `\\`, following kinds of names can be used:

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td><code>name</code></td><td>Name in standard language. Without quotation marks the name needs to be begin with a letter, an asterisk or with an underscore sign. Whitespaces or special characters which are used in other expressions are not allowed. The name must comply with following regular expression:

<code>[a-zA-Z_*][^/(){}$%{}[],~@§#+-'"s|^&]*</code>

For better readability, the escape character <code>\</code> has been left out</td></tr>
<tr><td><code>"name"</code> or <code>'name'</code></td><td>If the name doesn’t meet the above-mentioned requirements, it needs to be surrounded by single quotes or double quotes. Here, the backslash sign <code>\</code> serves as escape character for possibly used apostrophes, e. g. <code>Wendy's</code>.</td></tr>
<tr><td><code>nameen</code></td><td>Name in the specified language "lang"</td></tr>
<tr><td><code>$name$</code>, <code>$"name"$</code></td><td>Internal name</td></tr>
<tr><td><code>§name§</code>, <code>§"name"§</code></td><td>System name</td></tr>
<tr><td><code>#ID42_1013</code></td><td>ID of the object</td></tr>
</table>

Names are not replaceable by variables and must therefore be a part of the script.

1.3.3.5.2. Operators

Numeric values can be linked by the operators `+`, `-`, `*` or `/`.

When using `*`, `-` and `/`, at least one white space character must surround the operator on both sides each.

Parenthesis are supported, e. g. `(5 + 3) * 4` equals the value 32.

Sum of all relations between Goethe and Schiller:

```
\\Goethe/~*/size() + \\Schiller/~*/size()
```

The operator + also can be used to append strings:

```
//person\Goethe + " wrote " + //book\Faust
```

leads to:

```
Goethe wrote Faust
```

By means of the unary operator !, a Boolean expression can be negated, e. g.:

```
!1=2
```

For some operators, an alternative notation only consisting of alphabetical characters is possible, e. g. eq for equality. Applying this notation, at least one white space character needs to be used between operator and operand. The expressions are case-sensitive, so operators are only recognized if written in small letters.

Possible operators are (in descending precedence):

Operator	Alternative notation	Meaning
!	not	Negation (unary operator)
*		Multiplication
/		Division
+		Addition, linking (only character strings)
-		Subtraction
<	lt	Smaller than
>	gt	Greater than
⍰	le	Smaller than or equal to
>=	ge	Greater than or equal to
=	eq	Equal to
!=	ne	Not equal to

Operator	Alternative notation	Meaning
^^	xor	Exclusive or (logical operator)
&&	and	And (logical operator)

Due to KScript basing on XML, operators like &&, < or <= need to be written using entities like < or & instead of the character signs < and & or alternative notation needs to be used.

Example for "and":

```
<Path path="var(left) &amp;&amp; var(right)"/>
<Path path="var(left) and var(right)"/>
```

Example for "smaller than":

```
<Path path="var(left) &lt; var(right)"/>
<Path path="var(left) lt var(right)"/>
```

1.3.3.5.3. Conditions

Conditions can be specified using the following notation:

```
path1[path2]path3
```

On all elements out of path1 for which the condition path2 applies, path3 will be executed. To express the condition path2, comparative operators can be used (see preceding section). Boolean expressions can be linked with Boolean operators.

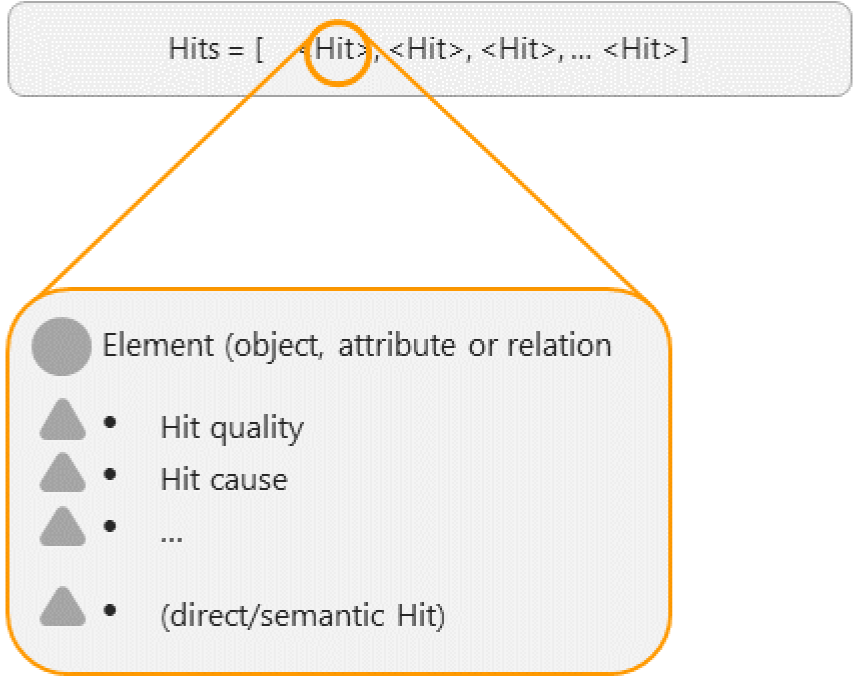
Name of all books which had been written by Goethe:

```
//book\*[~author/target()/@Name = "Goethe"]/@Name/value()
```

1.3.4. Model "Hit"

The "Hit" type content model is available to ensure that search queries can be processed and transported both as quality and causes. A "Hit" can be seen as a container that summarizes the element including several properties and makes it temporarily available to the context. The contained properties can be, for example, calculated hit quality, hit cause, change log entry etc.

In search pipelines, the content models "Hit" and "Hits" are available. The "Hits" type is an array of several "Hit" elements:



1.3.4.1. Meta-attributes of hits

In addition to the semantic element, the following meta-attributes are transported in a hit:

Hit quality

Can have a value between 0 and 1 by setting a quality in a search pipeline; the hits of a structured query receive the value 1 by default

Hit cause

Refers to the input element that has led to the hit and its type

Hit cause (snippet)

Refers to the content or the search term that has led to the hit

For detailed information on the meta-attributes, refer to the JavaScript API.

1.3.4.2. Using hits in search pipelines

If a hit list is to be processed in a search pipeline by means of a simple query, individual processing is required because the hit list is in the form of an array: Queries can process an individual "hit" in the form of a string but not "hits" (= array). Converting a "hit" into string, in turn, can be done using a script that precedes the simple query.

Process individually hits => hit
- Script hit => name
- Query name => elements
- Merge partial hits elements => results

Example script for converting a hit into a string:

```
function search(input, inputVariables, outputVariables) {
  return input.element().name();
}
```

1.3.4.3. Using hits in tables

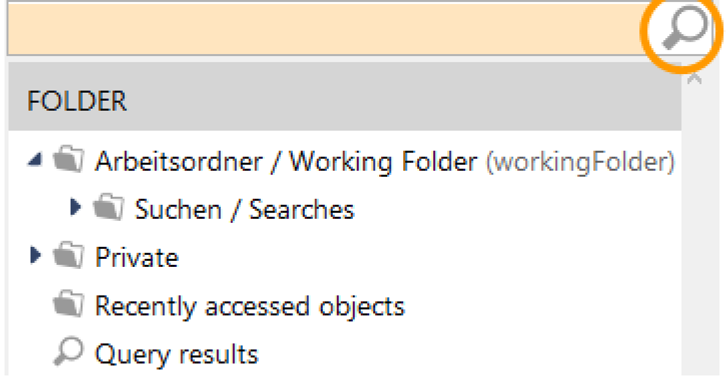
The *Use hits* option is available in the query configuration above the table. This option determines whether the entire hit element (semantic element + meta-attributes) or only the semantic element is to be forwarded to display query results.

1.3.4.4. Processing hits in tables via a script

If the query results are to be processed further using a script, the *Use hits* option determines whether the query result is supposed to be treated as a hit: The script is forwarded either $k.SemanticElement or $k.Hit as a JavaScript object.

1.3.5. Search in the Knowledge Builder

With the exception of the structured queries which are created in the folders and also implemented there, all searches in the header of the Knowledge Builder are made available for internal usage.



For this purpose we only have to drag & drop a pre-configured search into the search box of the header of the Knowledge Builder. If this contains several searches to be selected from you can select the desired search from the pull-down menu by clicking on the magnifier icon . The search input box always preserves the search mode which was last executed.

We can remove the search using the global settings where we can also change the sequence of the various searches in the menu.

1.3.6. Special cases

1.3.6.1. Fulltext search with Lucene

The full-text search may also alternatively be carried out via the external indexer *Lucene*. The search configuration is then analogous to the standard full-text search, i.e. attributes may, in turn, be configured in the search which are also connected to the Lucene index; the search process is also analogous. In order to configure the Lucene indexer connection we hereby refer you to the corresponding chapter in the admin manual.

1.3.6.2. Search with regular expressions

Regular expressions are a powerful means of searching through databases for complex search expressions, depending on the task concerned.

Search with regular expressions	hit
The [CF]all	The Call, The Fall
Car.	Cars
Car.*	Cars, Caravans, Carmen, etc.
[^R]oom	doom, loom, etc. (but not room)

As search inputs, i-views supports the standard also known from the standard known from Perl which, for example, is described in the Wikipedia article for regular expression.

1.3.6.3. Search in folders

The search in folders is carried out in names of folders and their contents:

- folders whose name matches the search input
- folders which contain objects which match the search input
- expert searches which contains elements which match the search input
- scripts in which the search input appears
- rights and trigger definitions which contain elements which match the search input

Using the search input "#obsolete", you can target your search for deleted objects (e.g. searching in rights and triggers). When configuring the search the number of folders to be searched through can be limited. Furthermore, the option "search for object names in folders" may be deactivated. This is helpful if you do not want to search for semantic objects in folders because in the case of extensive folders (e.g. saved search results) the search for object names may take a very long time.

1.3.6.4. Query for duplicates

After imports or due to other reasons such as quality assurance it can be necessary to check for duplicates semantic elements. To do so, a specially configured structured query can be used.

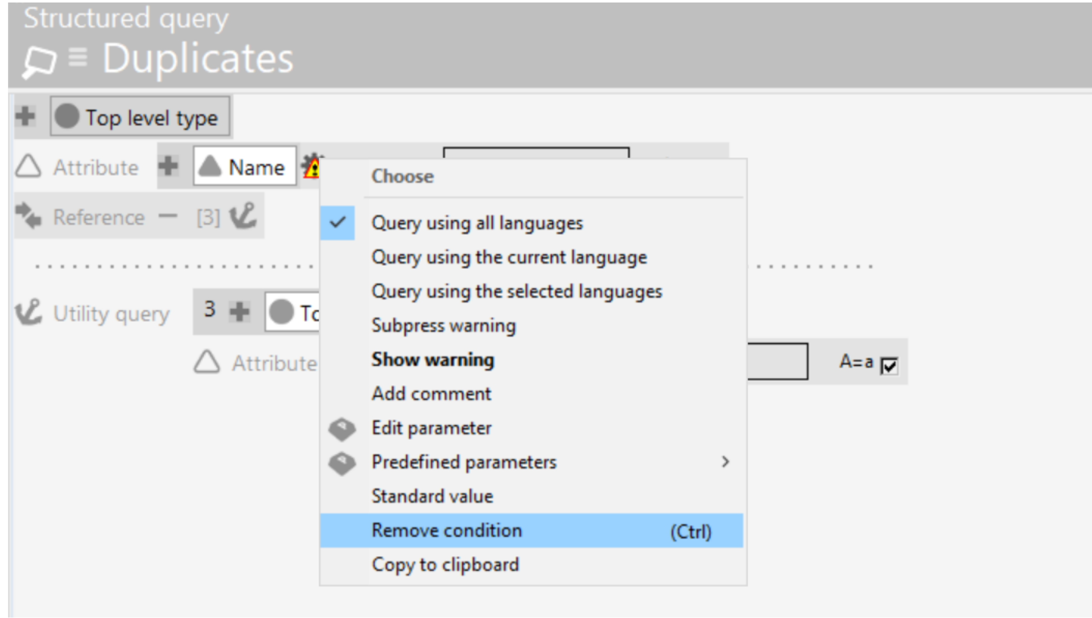
NOTE Because the structured query shown in the following example refers to elements of the whole Knowledge Graph without further type restrictions (objects of top level type), executing the query can take very long. It therefore is advised to restrict the query to the most specific subtype possible.

In principle, the structured query searches for **different** objects that have identical values for their identifying attributes (here: objects with identical names).

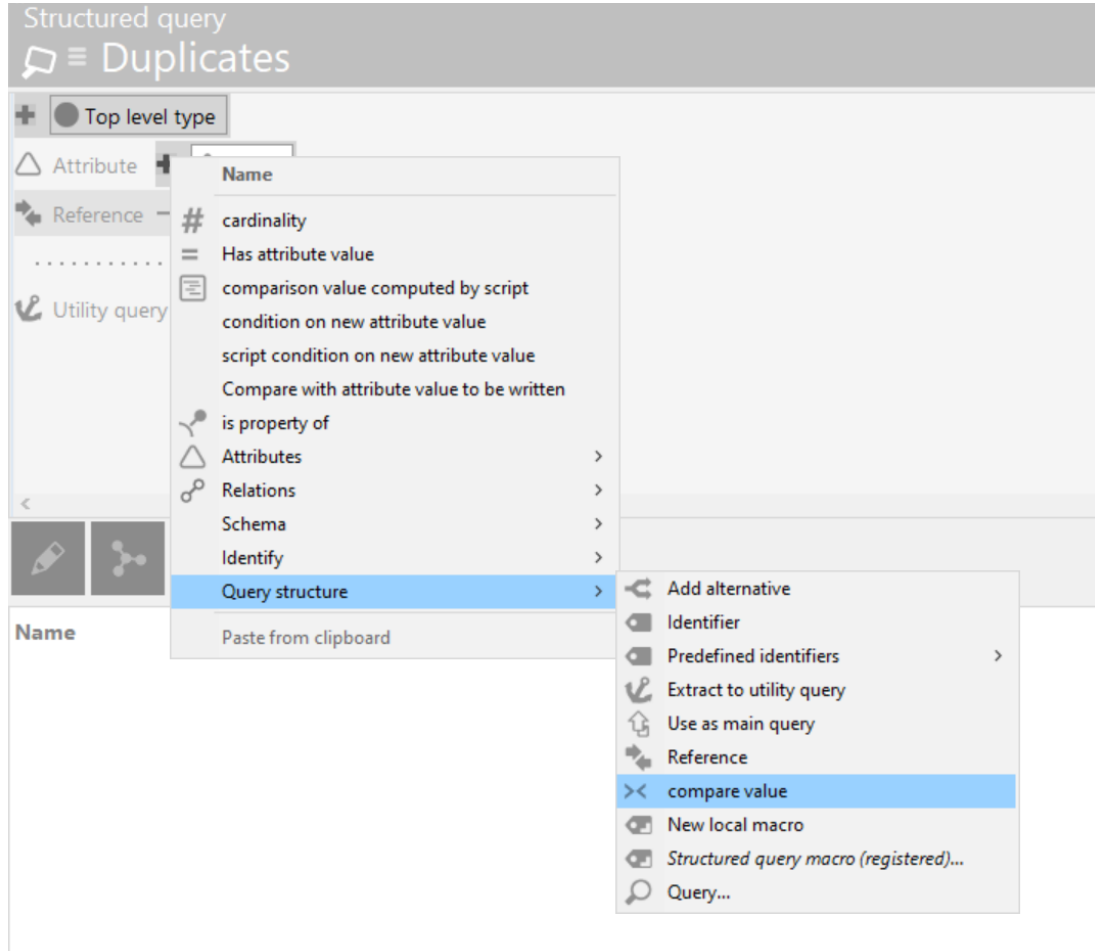
The query for duplicates can be configured as follows:

1. Create a query for objects of the subtype in question. Add the identifying attribute as condition (here: primary name).
2. Depending on the object, create a utility query. Use a negative reference (comparison operator "is not") to make sure that only different objects will be found:

3. For comparing the attribute values against each other, the value fields need to be removed first:



4. After having removed the value fields, the context menu offers the option "compare values". Add this condition and select the identifying attribute of the utility query to compare against:



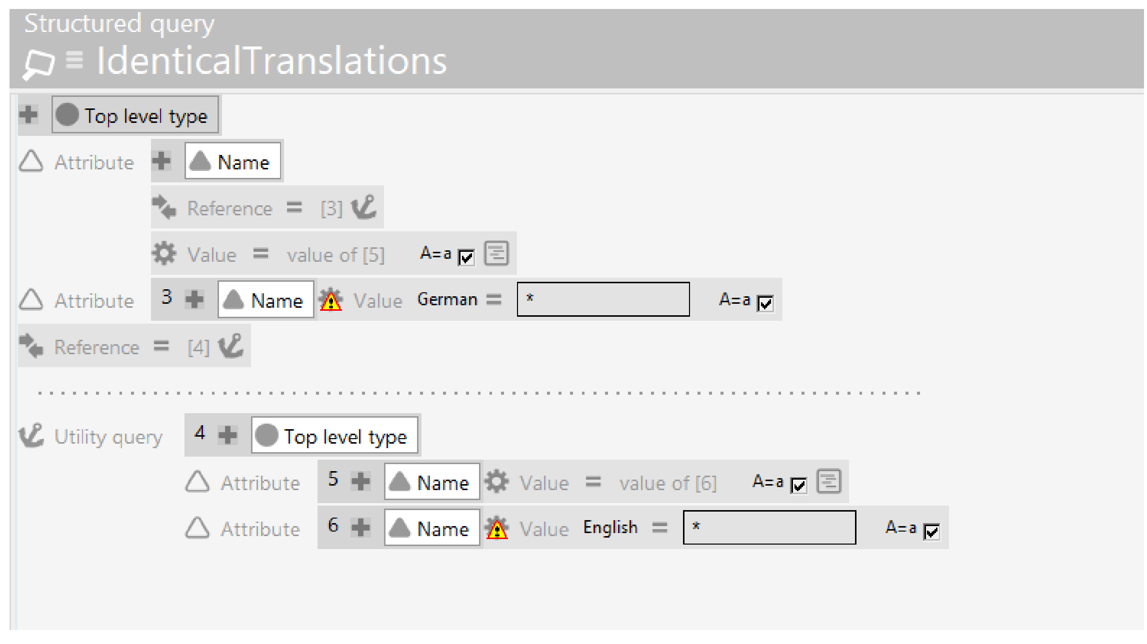
Result: Structured query for searching duplicates.

1.3.6.5. Query for identical translations

Similar to the query for duplicates, the query for objects with identical translations makes use of references and attribute value comparisons.

NOTE	Because the structured query shown in the following example refers to elements of the whole Knowledge Graph without further type restrictions (objects of top level type), executing the query can take very long. It therefore is advised to restrict the query to the most specific subtype possible.

The difference between this query and the query for duplicates is that it is all about one and the same object this time, with the additional condition of identical attribute values in different translation layers of one and the same attribute:



1.3.7. Graph Query Language (GQL)

The Graph Query Language (GQL) is an ISO standard for querying and modifying graph databases, modelled after the proprietary language Cypher.

GQL allows you to formulate patterns for nodes, edges, and properties to search for in a compact way. The result consists of a list of fields containing properties or calculated values.

Website: https://www.gqlstandards.org

NOTE Data-modifying statements are not yet supported.

NOTE The GQL standard is young and few reference implementations exist yet. It is therefore possible that the behaviour of the i-views implementation deviates from the standard in individual points. Future revisions of the standard may also introduce changes to the prescribed behaviour. In such cases the standard is authoritative, and deviating behaviour may be corrected at any time — for example in patch releases — without prior notice.

This does not apply to i-views-specific language extensions, as these are outside the scope of the GQL standard.

1.3.7.1. Structure

The following query finds all manufactured products whose weight is at least 10, and returns manufacturer, product, and weight sorted by weight:

```
MATCH (m:manufacturer)-[:produces]->(p:product)
WHERE p.weight >= 10
RETURN m.name as manufacturer, p.name as product, p.weight as weight
ORDER BY weight
```

Result

manufacturer	product	weight
ACME	Artificial Rock	50
ACME	Anvil	230

`MATCH` lists one or more paths to be searched for in the graph. A path alternates between a **node** in round brackets and an **edge** in square brackets. The beginning and end must be a node.

Nodes are typically objects in the knowledge graph, and edges correspond to relations.

Nodes and edges consist of an optional variable and an optional label, separated by a colon. In

`m:manufacturer`, `m` is the variable and `manufacturer` is the label. For the edge `[:produces]`, the variable was omitted.

The label specifies the type of semantic elements to search for. By default, the label is the internal name of the type. You can also set up a custom label configuration.

Using the variable, you can reference the node or edge again later in the query.

`WHERE` specifies additional conditions for the nodes and edges to be found. The expression `p.weight >= 10` references the product nodes via the variable `p`. `weight` is the label of a property of the products.

Conditions can only reference nodes and edges listed in the MATCH section; no additional nodes or edges can be queried.

`RETURN` specifies a list of elements to be included in the result. In addition to properties, values can also be calculated using expressions. For each element listed in RETURN, one field is output in the result. The name of the field can be determined using `AS name`. The result corresponds to a table with one column per field and one row per matching assignment of the fields.

The optional `ORDER BY` sorts the results. You can also use properties/values that are not part of RETURN.

1.3.7.2. Comparison with structured queries

- In contrast to structured queries, GQL searches are formulated as text. Schema support is minimal and is typically only evaluated at runtime.
- The result does not consist of an unsorted set of semantic elements, but of a list of assignments of the field elements specified in `RETURN`. The list can be sorted using `ORDER BY`.

1.3.7.3. Executing GQL searches

There are several ways to execute GQL queries in the knowledge graph:

- Workbench
- GQL searches as folder elements or registered searches
- REST interface
- JavaScript API

1.3.7.3.1. Workbench

In the Workbench you can formulate and execute any GQL searches. You open the Workbench in the main menu under ***Tools > GQL Workbench***.

1.3.7.3.2. GQL searches as folder elements or registered searches

When creating a new search as a folder element or registered search, you can select ***GQL Query*** from the list in the dialog.

You can then execute the search like other types of searches (structured query, etc.). GQL differs from other search types in that GQL returns a table as its result, while other searches return a set of semantic elements or hit objects.

To be compatible with other searches, GQL searches called via the standard APIs/tools return the variable assignments from the returned rows collected together.

For the search

```
MATCH (m:manufacturer)-[:produces]->(p:product)-[:contains]->(c:component)
WHERE c.weight > 1000
RETURN m.name, p.name
```

manufacturers (`m`) and products (`p`) are returned as the result.

Access to the result table is only possible using the dedicated GQL API.

1.3.7.4. Path Patterns

1.3.7.4.1. Nodes

In the simplest case, the path pattern in `MATCH` consists of a single node. Nodes consist of round brackets `()` containing an optional variable and an optional label.

If a label is specified, a colon `:` must precede the label, even if no variable is given: `(:company)`.

The following pattern assigns to the variable `m` all nodes with the label `manufacturer`. The label corresponds to a type in the knowledge graph, and the nodes encompass all objects of that type and all subtypes.

```
MATCH (m:manufacturer) RETURN m
```

1.3.7.4.2. Labels

The label typically corresponds to the internal name of a type in the knowledge graph.

Labels can contain any characters. However, they must be enclosed in double quotes (`"`) or backticks (`` ` ``) if they do not consist exclusively of letters, digits, or underscores (`_`). The first character must not be a digit. This corresponds to a Unicode identifier. The single quote `'` cannot be used.

```
MATCH (p:"base.product") RETURN p
```

Labels can be combined:

- Union: `MATCH (:corporation|association)`
- Intersection: `MATCH (:product&cataloged)`
- Exclusion, normally only used in combination with intersection: `MATCH (:company&!association)`

The label is optional and can be omitted. Only omit it if the set of nodes/edges is restricted by further path patterns. In the following query, the set of nodes `m` is already restricted by the edge `:produces`:

```
MATCH (m)-[:produces]->(p:product) RETURN m, p
```

The following query without a label addresses the entire knowledge graph and should therefore be avoided:

```
MATCH (m) RETURN m
```

1.3.7.4.3. Edges

Edges connect nodes in the path pattern and correspond to relations in the knowledge graph. Edges are represented by square brackets `[]`; otherwise the same rules apply as for nodes. Edges can also be assigned to a variable and returned.

The following pattern finds all relations of the relation type with the internal name `produces` and returns them:

```
MATCH (:company)-[p:produces]->(:product) RETURN p
```

The label is optional. However, do not omit it for edges. Without a label, all relations between the nodes are determined, including system relations and shortcut relations — this is very costly, even when the nodes have labels.

Edges in GQL can be undirected or directed. For directed edges, `<` or `>` precedes the target node. For undirected edges, the path contains no angle brackets.

It is also permitted to specify both directions, which makes no difference in the GQL implementation in the knowledge graph.

Pattern for "*Component c1 is part of c2*"

```
MATCH (c1:component)-[:partOf]->(c2:component) RETURN c1, c2
```

Pattern for "*Component c2 is part of c1*"

```
MATCH (c1:component)<-[:partOf]-(c2:component) RETURN c1, c2
```

Pattern for "*Component c1 is part of or contains part c2*", formulated once as an undirected edge and once as an edge in both directions

```
MATCH (c1:component)-[:partOf]-(c2:component) RETURN c1, c2
```

```
MATCH (c1:component)<-[:partOf]->(c2:component) RETURN c1, c2
```

Edges connect nodes that must always be specified in the path. At minimum, an empty pair of round brackets is sufficient: `()`.

```
MATCH ()-[p:produces]->() RETURN p
```

Paths can contain multiple edges:

```
MATCH (c:component)-[:partOf]->(p:product)<-[:produces]-(m:manufacturer)
RETURN c,p,m
```

1.3.7.4.4. Property filters in patterns

In addition to the label, nodes and edges in the pattern can be given property conditions directly. This is a compact alternative to the `WHERE` clause.

Property filters with record syntax

A record in curly brackets after the label specifies that the given properties must match the given values exactly.

```
MATCH (p:product {category: 'electronics', inStock: TRUE})
RETURN p.name, p.price
```

The same syntax applies to edges:

```
MATCH (p:person)-[:knows {since: 2020}]->(q:person)
RETURN p.name, q.name
```

Inline WHERE clause

For more complex conditions, a WHERE clause can be specified directly inside the node or edge pattern. This allows arbitrary expressions, not only equality comparisons.

```
MATCH (p:product WHERE p.price < 100 AND p.rating >= 4)
RETURN p.name, p.price, p.rating
```

```
MATCH (p:person)-[r:knows WHERE r.since > 2015]->(q:person)
RETURN p.name, q.name, r.since
```

1.3.7.4.5. Patterns with multiple paths

A path is always a linear sequence of nodes and edges, meaning there are no branches. To formulate more complex patterns, multiple paths can be specified. Paths can reference nodes and edges from previous paths.

Reusing variables

When the same variable is used in multiple paths, it is always bound to the same element. This means that all paths using the variable must refer to the same node or edge object. In this way you can express conditions across multiple paths.

```
MATCH (p:person)-[:owns]->(t:product),
      (p)-[:worksAt]->(c:company)
RETURN p.name, t.name, c.name
```

The variable p appears in both paths and ensures that the product and the company belong to the same person. Without this link, every combination of person, product, and company would be returned.

Example with multiple linked paths

The following query finds combinations of component, product, manufacturer, and qualification for which:

- The component is part of the product

- The product is produced by the manufacturer
- The product requires the qualification

```
MATCH
  (c:component)-[:partOf]->(p:product)<-[:produces]-(m:manufacturer),
  (p)-[:requires]->(q:qualification)
RETURN
  c.name as component, p.name as product, m.name as company, q.name as
qualification
```

The variable p in the second path references the product p from the first path.

If there are multiple components, manufacturers, products, or qualifications, all combinations are listed individually:

component	product	company	qualification
Compressor	Jet Motor	ACME	Technician
Compressor	Jet Motor	ACME	Pilot
Turbine	Jet Motor	ACME	Technician
Turbine	Jet Motor	ACME	Pilot

1.3.7.4.6. Quantified paths

Path patterns can be given a quantifier to search for paths of variable length. The quantifier is placed in curly brackets after the pattern element to be repeated and specifies the number of allowed repetitions. Each individual node or edge element can be quantified, as can entire sub-paths in round brackets.

Quantifier	Meaning
`{n}`	Exactly n repetitions
`{n,m}`	At least n, at most m repetitions
`{n,}`	At least n repetitions (open-ended)
`{,m}`	At most m repetitions

```
MATCH (p:person)-[:knows]->{1,3}(q:person)
RETURN p.name, q.name
```

This pattern finds persons p and q that are connected by one to three knows edges. The edge :knows is quantified; the intermediate nodes implicitly created in the process are anonymous. The

explicitly specified target node `(q:person)` is not part of the quantification and always designates a single node.

Entire sub-paths in round brackets can also be quantified:

```
MATCH (p:person)(-[:owns]->(:item)<-[:wants]-(:person)){1,3}(q:person)
RETURN p.name, q.name
```

This pattern finds persons `p` that are connected to a target person `q` by one to three steps of "owns an item that a person wants".

NOTE	For open-ended quantifiers without an upper bound (`{n,}`), you must ensure that the graph contains no cycles or that the search is otherwise restricted, as otherwise very long paths will be searched.

Group variables

Variables defined within the quantified part of a pattern are available outside the pattern as group variables. A group variable contains a list of all values that were bound across all repetitions of the pattern. All list functions can be applied to it.

```
MATCH (p:person)-[r:knows]->{1,3}(q:person)
RETURN p.name, q.name, SIZE(r) AS hops
```

Here `r` contains a list of all `knows` edges along the path, and `SIZE(r)` returns the number of hops.

1.3.7.4.7. Optional paths

A path pattern can be marked as optional with `?`. The pattern is found either once or not at all. As with quantified paths, each individual element or a bracketed sub-path can be made optional.

In contrast to quantified paths, `?` does not create group variables. Variables in the optional part are single values that are `NULL` if the pattern is not found — the result row itself is retained.

```
MATCH (p:person)-[r:manages]->?(q:person)
RETURN p.name, q.name, r
```

If no `manages` edge exists, `r` contains the value `NULL` and `p` and `q` refer to the same person.

1.3.7.5. Values

The following literal value types are defined by the GQL standard:

Strings

Strings are enclosed in single or double quotation marks: `'Some string'` or `"Another string"`.

> **NOTE** The quotation mark itself can be included in the string by preceding it with a backslash: `"A \"quote\""`

Numbers

Possible notations are

- integers, (`42`)
- decimal numbers (`-1.23`)
- floating-point numbers with mantissa and exponent and optional precision (`-1.4e6`, `2.34e9d`)
- hexadecimal, octal, and binary integers (`0x1f`, `0o7`, `0b101`)

> **NOTE** The knowledge graph only supports integers and double-precision floating-point numbers; all other formats are converted.

Null

The value `NULL` represents a non-existent value, but can also be used as a literal value.

Boolean values

In addition to `TRUE` and `FALSE`, GQL uses `NULL` as an unknown boolean value.

> **WARNING** GQL additionally defines `UNKNOWN` as a keyword for an unknown boolean value. `UNKNOWN` is, however, not a valid literal value and can only be used with the `IS [NOT]` operator.
>
> However, `NULL` can also be used here: `IS [NOT] NULL` is equivalent to `IS [NOT] UNKNOWN`. You can therefore omit the use of `UNKNOWN`.

Date/time

GQL defines date, time, and date and time (also called timestamp) values. Values with a time component can be specified either with or without a time zone.

GQL follows the standards `ISO 8601-1:2019` and `ISO 8601-2:2019` for its syntax.

Value type	Notation	Value
Date	`DATE "2025-02-28"`	February 28, 2025
Date	`DATE "20240229"`	February 29, 2024

Value type	Notation	Value
Time	`TIME "12:34:56"`	12:34:56
Time	`TIME "23:59"`	23:59:00
Date and time	`DATETIME "2025-02-23T13:45"`	2/23/2025 13:45:00
Date and time	`TIMESTAMP "2025-03-05T02:03:04"`	3/5/2025 02:03:04

NOTE The knowledge graph does not support time zones; times specified with a time zone are converted to local time.

Durations

In GQL, durations can be specified at the date or time level.

GQL follows the standards `ISO 8601-1:2019` and `ISO 8601-2:2019` for its syntax.

Notation	Duration
`DURATION "P1Y"`	1 year
`DURATION "P2M"`	2 months
`DURATION "P30D"`	30 days
`DURATION "P1Y2M"`	1 year 2 months
`DURATION "PT4H"`	4 hours
`DURATION "PT120M"`	120 minutes
`DURATION "PT30S"`	30 seconds

Lists

Lists are ordered sequences of values of any type. A list is enclosed in square brackets and the elements are separated by commas.

```
['ACME', 'Mustermann GmbH']
[1, 2, 3]
[DATE "2025-01-01", DATE "2025-06-01"]
```

An empty list is written as `[]`. The elements of a list do not all have to be of the same type.

NOTE Accessing list elements via an index is not supported in GQL.

Records

Records are collections of named fields. A record is enclosed in curly braces; each field consists of a key, a colon, and a value; multiple fields are separated by commas.

```
{companyName: 'ACME'}
{searchString: name, maxResults: 10}
{distanceValue: '100km'}
```

Keys follow the same rules as property names: they must be enclosed in double quotation marks or backticks if they do not consist exclusively of letters, digits, or underscores.

1.3.7.6. Expressions

Conditions, values to be returned, and values for sorting are formulated using expressions.

NOTE In the following, identifiers enclosed in angle brackets such as `<value>` denote an expression.

1.3.7.6.1. Literal Values

Every valid literal value is also an expression. See Values.

1.3.7.6.2. Variable References

Variables of the path pattern can be referenced by their name. The result of the expression is the binding of the variable, i.e. a node, edge, or path.

```
MATCH p1 = (m:manufacturer)-[r:produces]->(p:product)
RETURN m, r, p, p1
```

In the example, `m`, `r`, and `p` are variable references pointing to the manufacturer node, the edge, and the product node. `p1` is a path variable that contains the entire path.

NOTE `path` is a reserved keyword in GQL and cannot be used as a variable name.

1.3.7.6.3. Properties

Properties correspond to attributes in the knowledge graph.

Analogous to labels, property names normally correspond to the internal names of attribute types. Names must likewise be enclosed in double quotation marks (`"`) or backticks (`` ` ``) if they do not consist exclusively of letters, digits, or underscores (`_`).

NOTE Unlike labels, property names cannot be combined with `&`, `|`, or `!`.

If a node or edge does not have the specified property, `NULL` is returned as the value.

```
MATCH (m:manufacturer)-[:produces]->(p:product)
RETURN m.name, m.isin, p.name, p.id
```

1.3.7.6.4. Operators

The result of an operator depends on the type of the values involved.

General Operators

Operator	Result
`+`	Adds numeric or chronological values.
`-`	Subtracts numeric or chronological values.
`*`	Multiplies numeric values or durations
`/`	Divides numeric values or durations
`||`	Concatenates lists or strings

Comparison Operators and Predicates

Comparison operators return a boolean value. For binary operators, the values must be comparable.

NOTE	Currently the compared values must be of the same type; the GQL feature *GA04 Universal comparison* is not supported.

Operator	Result
`=`	Equal
`<>`	Not equal
`<, <=, >, >=`	Less than (or equal), greater than (or equal)
`<boolean> IS [NOT] <truth value>`	Compares a boolean value with a truth value (`TRUE`, `FALSE`, `UNKNOWN`). `UNKNOWN` is equivalent to a `NULL` value (see Expressions with Null Values).
`<value> IS [NOT] NULL`	True if the value is (not) a `NULL` value
`<node_or_edge> IS [NOT] LABELED <label>`	Checks at runtime whether a node or edge has a specific label.
`PROPERTY_EXISTS(<node_or_edge>, <identifier>)`	Checks whether a node or edge has the specified property.

Operator	Result
`SAME(<element1>, <element2>, …)`	Checks whether all specified variables refer to the same graph element.
`ALL_DIFFERENT(<element1>, <element2>, …)`	Checks whether all specified variables refer to pairwise distinct graph elements.

Logical Operators

The following logical operators can only be applied to boolean values.

Operator	Result
`AND`	Boolean AND
`OR`	Boolean OR
`XOR`	Boolean exclusive OR
`NOT`	Boolean negation

NOTE Unlike JavaScript, GQL has no concept of "truthy" or "falsy"; other values such as numbers are therefore not implicitly converted to boolean values.

1.3.7.6.5. EXISTS Expression

You can use `EXISTS { … }` to check whether a pattern exists in the graph without returning the matched elements. The result is a boolean value and can be used directly in a `WHERE` or `FILTER` clause. `NOT EXISTS { … }` checks whether no matching pattern exists.

```
MATCH (p:product)
WHERE EXISTS {
  MATCH (p)-[:contains]->(c:component)
  WHERE c.weight > 10
}
RETURN p.name
```

This pattern returns all products that contain at least one component with a weight greater than 10.

```
MATCH (p:person)
WHERE NOT EXISTS {
  MATCH (p)-[:owns]->(:product)
}
RETURN p.name
```

This pattern returns all persons who do not own any product.

NOTE `EXISTS` is an alternative to `OPTIONAL MATCH` in cases where only the presence of a pattern is relevant and no properties of the matched elements are needed.

1.3.7.6.6. CASE Expression

A `CASE` expression allows you to return different values depending on conditions. A `CASE` expression consists of one or more `WHEN` clauses and an optional `ELSE` branch. If no `WHEN` clause matches and no `ELSE` is specified, the result is `NULL`.

Simple Form

In the simple form, an expression is compared for equality against a list of values.

```
MATCH (p:product)
RETURN p.name,
  CASE p.category
    WHEN 'electronics' THEN 'electronics'
    WHEN 'furniture' THEN 'furniture'
    ELSE 'other'
  END AS categoryLabel
```

The simple form can also be used with a predicate. The expression after `CASE` then serves implicitly as the left-hand side of the predicate.

```
MATCH (p:product)
RETURN p.name, p.price,
  CASE p.price
    WHEN < 10 THEN 'cheap'
    WHEN < 100 THEN 'medium'
    WHEN IS NULL THEN 'unknown'
    ELSE 'expensive'
  END AS priceCategory
```

Searched Form

In the searched form, each `WHEN` clause is evaluated as an independent boolean condition. This allows arbitrary expressions, not just comparisons of a single value.

```
MATCH (p:product)
RETURN p.name,
  CASE
```

```
    WHEN p.price < 10 AND p.rating >= 4 THEN 'Recommendation'
    WHEN p.price >= 10 AND p.rating >= 4 THEN 'Good, but expensive'
    ELSE 'No recommendation'
  END AS recommendation
```

1.3.7.6.7. Functions

Functions are part of the GQL syntax; there is no generic function call mechanism. As a result, some functions differ syntactically from the remaining functions.

NOTE If a function argument has the value `NULL`, the result of the function is also `NULL`. This applies only to passed arguments — optional arguments that are omitted do not trigger this behavior.

Mathematical Functions

Function	Result
`ABS(<value>)`	Absolute numeric value
`FLOOR(<value>)`	Largest integer less than or equal to the argument
`CEIL(<value>)`	Smallest integer greater than or equal to the argument
`MOD(<dividend>, <divisor>)`	Modulus (remainder of integer division)
`SQRT(<value>)`	Square root
`POWER(<base>, <exponent>)`	Power
`EXP(<value>)`	Exponential function (e to the power of `<value>`)
`LN(<value>)`	Natural logarithm
`LOG(<base>, <value>)`	Logarithm to the specified base
`LOG10(<value>)`	Decimal logarithm
`SIN(<value>)`, `COS(<value>)`, `TAN(<value>)`, `COT(<value>)`	Trigonometric functions (argument in radians)
`ASIN(<value>)`, `ACOS(<value>)`, `ATAN(<value>)`	Inverse trigonometric functions, result in radians
`SINH(<value>)`, `COSH(<value>)`, `TANH(<value>)`	Hyperbolic functions

Function	Result
DEGREES(<value>), RADIANS(<value>)	Conversion between radians and degrees

String Functions

Function	Result
UPPER(<string>)	String in uppercase
LOWER(<string>)	String in lowercase
NORMALIZE(<string> [, NFC|NFD|NFKC|NFKD])	Unicode normalization of the string into the specified normal form (default: NFC).
TRIM(<string>)	String without leading and trailing whitespace. Optional arguments control which characters are removed and from which side: TRIM(LEADING '#' FROM <string>), TRIM(TRAILING FROM <string>), TRIM(BOTH '.' FROM <string>). Without specification, BOTH applies and whitespace is used as the trim character.
LTRIM(<string> [, <chars>]), RTRIM(<string> [, <chars>]), BTRIM(<string> [, <chars>])	String without leading (LTRIM), trailing (RTRIM), or both-sided (BTRIM) occurrences of the specified string. Without specifying <chars>, whitespace is removed.
CHAR_LENGTH(<string>)	Length of the string in characters
LEFT(<string>, <n>)	The first <n> characters of the string
RIGHT(<string>, <n>)	The last <n> characters of the string

List Functions

Function	Result
SIZE(<list>)	Number of elements in the list
TRIM(<list>, <n>)	List of the first <n> elements of the list

Graph Functions

Function	Result
ELEMENT_ID(<node_or_edge>)	Internal ID of the node or edge as a string
PATH_LENGTH(<path>)	Number of edges in the path
ELEMENTS(<path>)	Elements of the path as a list (nodes, edges, and sub-paths)

Date/Time Functions

<table>
<tr><th>Function</th><th>Result</th></tr>
<tr><td>CURRENT_DATE, DATE()</td><td>Current date</td></tr>
<tr><td>DATE(<string>)</td><td>Date from a string (ISO 8601 format)</td></tr>
<tr><td>DATE({year} | {year, month} | {year, month, day})</td><td>Date from individual components</td></tr>
<tr><td>CURRENT_TIME, LOCAL_TIME()</td><td>Current time</td></tr>
<tr><td>LOCAL_TIME(<string>)</td><td>Time from a string (ISO 8601 format)</td></tr>
<tr><td>LOCAL_TIME({hour} | {hour, minute} | {hour, minute, second} | {hour, minute, second, millisecond|microsecond|nanosecond})</td><td>Time from individual components</td></tr>
<tr><td>CURRENT_TIMESTAMP, LOCAL_DATETIME()</td><td>Current date and time</td></tr>
<tr><td>LOCAL_DATETIME(<string>)</td><td>Date and time from a string (ISO 8601 format)</td></tr>
<tr><td>LOCAL_DATETIME({year, month, day} + any valid time components)</td><td>Date and time from individual components</td></tr>
<tr><td>DURATION(<string>)</td><td>Duration from a string (ISO 8601 format)</td></tr>
<tr><td>DURATION({year} | {year, month} | {year, month, day} | {hour} | …)</td><td>Duration from individual components</td></tr>
</table>

Function	Result
`DURATION_BETWEEN(<value1>, <value2>)`	Duration between two comparable date/time values
`ABS(<duration>)`	Absolute value of a duration

NOTE

Aggregation functions such as `COUNT`, `SUM`, `AVG`, etc. are available in `RETURN` expressions and are described in Returning Values.

i-views-specific functions (`IV_COMPARE`, `IV_VALUE`, `IV_ACCESS_PARAMETER`) are described in i-views-Specific Language Extensions.

1.3.7.6.8. Expressions with Null Values

The result of functions or operators is `NULL` in almost all cases when one of the arguments is `NULL`. In the addition `p.netWeight + p.packagingWeight`, for example, the result is `NULL` if the value `netWeight` is `NULL`, regardless of the value of `packagingWeight`.

When a value is evaluated as a boolean, `NULL` corresponds to the value `FALSE`. With boolean operators, `NULL` is preserved; `NULL AND TRUE`, for example, evaluates to `NULL` and is treated as `FALSE`.

NOTE

If you want to check whether a value is `NULL`, you must not use the comparison operator, because the result of `NULL = NULL` is `NULL`.

Use the `IS` operator instead: `v IS NULL` or `v IS NOT NULL`.

Two special functions are also available for handling `NULL` values:

COALESCE

`COALESCE(<value1>, <value2>, …)`

Returns the first value from the argument list that is not `NULL`. If all arguments are `NULL`, the result is `NULL`. Useful for providing fallback values for missing properties.

```
MATCH (p:product)
RETURN p.name, COALESCE(p.salePrice, p.price) AS effectivePrice
```

NULLIF

`NULLIF(<value1>, <value2>)`

Returns `NULL` if both arguments are equal, otherwise the first value. Useful for treating specific values as missing.

```
MATCH (p:product)
RETURN p.name, NULLIF(p.status, 'discontinued') AS activeStatus

:leveloffset!:
:leveloffset: 4

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#gql-return]
= Returning Values

With `RETURN` you specify a list of values to return.
Each value is determined by an <<gql-expressions,expression>>.

== Returning Graph Elements

When returning path variables, the referenced graph elements (nodes,
```

```
edges, paths) are returned in JSON-like syntax.

Node:: Object with property `_id` containing the ID of the semantic
element
Edge:: Object with properties `_from` containing the ID of the source
and `_to` containing the ID of the target of the relation
+
[NOTE]
====
The ID of the relation is not included, because the ID is not contained
in some relation indexes and retrieving it would require additional
effort.
====
Path:: Object with properties `_nodes` (nodes of the path) and `_edges`
(edges of the path)

== Naming Fields

One field is added to the result per expression.
The name of the field defaults to the expression itself.
Alternatively, you can specify a different name using `AS`:

[source,cypher]
```

MATCH (c:component) RETURN c.id AS "Component ID"

```
The name must be enclosed in double quotes (`"`) or backticks (```) if it
does not consist exclusively of letters, digits, or underscores (`_`).

Field names must be unique; duplicates cause an error.

== Returning All Variables

With `RETURN *` all variables that were referenced in the `MATCH` part of
the query are returned.

[source,cypher]
```

MATCH (m:manufacturer)-[:produces]→(p:product) RETURN *

```
[%header,cols="a,a"]
|====
|m
|p
```

```
|{ _id: manufacturer-1 }
|{ _id: product-1 }

|{ _id: manufacturer-1 }
|{ _id: product-2 }

|====

== Duplicate Results

If not all nodes of the pattern are returned, or if the returned
properties of different nodes are equal, the result may contain
duplicates.

[source,cypher]
```

MATCH (c:component)-[:partOf]→(:product)←[:produces]-(m:manufacturer) RETURN c.name as component, m.name as company

```
[%header,cols="a,a"]
|====
|component
|company

|Compressor
|ACME

|Compressor
|ACME

|Turbine
|ACME

|Turbine
|ACME

|====

By using `RETURN DISTINCT` instead of `RETURN`, duplicate results can be
removed.

[source,cypher]
```

MATCH (c:component)-[:partOf]→(:product)←[:produces]-(m:manufacturer) RETURN DISTINCT c.name as component, m.name as company

```
[%header,cols="a,a"]
|====
|component
|company

|Compressor
|ACME

|Turbine
|ACME

|====

== Aggregation

Aggregation functions combine multiple rows of the intermediate result
into a single value.
They can be used in `RETURN` expressions.

[%header,cols="1m,3a"]
|===
| Function
| Result

| COUNT(<variable>)
| Number of rows in which the variable is not `NULL`

| COLLECT_LIST(<expression>)
| List of all values of the expression across all rows

| SUM(<expression>)
| Sum of all numeric values of the expression across all rows

| AVG(<expression>)
| Arithmetic mean of all numeric values of the expression across all rows

| MIN(<expression>)
| Smallest value of the expression across all rows

| MAX(<expression>)
| Largest value of the expression across all rows
```

```
| STDDEV_SAMP(<expression>)
| Sample standard deviation of the numeric values of the expression across
all rows

| STDDEV_POP(<expression>)
| Population standard deviation of the numeric values of the expression
across all rows

|===

=== Grouping with GROUP BY

Without `GROUP BY`, aggregation functions apply to all rows of the
intermediate result and return exactly one result row.

With `GROUP BY`, the rows are grouped by the specified expressions.
Aggregation functions are then evaluated per group and return one result
row per group.
`GROUP BY` is specified after `RETURN`.

[source,cypher]
```

MATCH (c:company)-[:produces]→(p:product) RETURN c.name AS company, COUNT(p) AS productCount GROUP BY c

```
[%header,cols="a,a"]
|====
|company
|productCount

|ACME
|3

|Mustermann GmbH
|1

|====

All expressions in `RETURN` that are not aggregation functions must be
listed in `GROUP BY` or must be derivable from the grouped variables.

[NOTE]
====
Since aggregation is only possible in `RETURN` expressions, you cannot
filter on aggregated values directly in a `WHERE` condition.
```

```
For this, a subsequent <<gql-composition-next,`NEXT`>> with `FILTER` must
be used (see <<gql-composition-next,Combining Queries>>).
====

== Sorting with ORDER BY

Result rows can be sorted with `ORDER BY`.
`ORDER BY` is specified after `RETURN` and contains a comma-separated list
of expressions to sort by.

[source,cypher]
```

MATCH (p:product) RETURN p.name, p.weight ORDER BY p.weight

```
You can also specify properties that are not part of `RETURN`.

[source,cypher]
```

MATCH (m:manufacturer)-[:produces]→(p:product) RETURN m.name AS manufacturer, p.name AS product ORDER BY m.name, p.name

```
=== Sort Direction

By default, sorting is ascending.
The direction can be specified explicitly with `ASC` or `ASCENDING`
(ascending) or `DESC` or `DESCENDING` (descending).
The direction applies to the preceding expression.

[source,cypher]
```

MATCH (p:product) RETURN p.name, p.price, p.weight ORDER BY p.price DESCENDING, p.name ASCENDING

```
=== NULL Values when Sorting

`NULL` values are treated as smaller than all other values by default when
sorting: with ascending sort they appear at the beginning, with descending
sort at the end.

This behavior can be controlled with `NULLS FIRST` or `NULLS LAST`.

[source,cypher]
```

MATCH (p:product) RETURN p.name, p.weight ORDER BY p.weight ASCENDING NULLS LAST

```
=== Pagination with LIMIT and OFFSET

With `LIMIT` you can restrict the number of returned rows.
With `OFFSET` you can skip a number of rows at the beginning of the
result.
Both are optional and are specified after `ORDER BY`, with `OFFSET`
required to come before `LIMIT`.

[source,cypher]
```

MATCH (p:product) RETURN p.name, p.price ORDER BY p.price OFFSET 20 LIMIT 10

```
This example returns rows 21 to 30 and is suitable for paginating through
results.

[NOTE]
====
`LIMIT` and `OFFSET` can also be used without `ORDER BY`.
However, the order of the results is then undefined, which is generally
unsuitable for pagination.
====

== Queries without Return Value

A query must end with RETURN.
For purely write queries where you do not need a return value, you can use
`FINISH` without further arguments instead of `RETURN`.

:leveloffset: 4
:leveloffset: 4

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none
```

```
// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#gql-composition]
= Combining Queries

In their simplest form, GQL queries consist only of

[source,cypher]
```

MATCH ... WHERE ... RETURN ...

```
For more complex queries, multiple MATCH conditions can be specified or
multiple result tables can be combined.

== Linear Queries

A linear query consists of a series of match conditions (`MATCH ...
WHERE`... , `FILTER`), followed by the return with `RETURN` and optional
sorting with `ORDER`.

=== Additional Conditions with FILTER

Conditions that you specify in the `WHERE` part can alternatively also be
specified in a `FILTER` statement.
In contrast to `WHERE`, multiple `FILTER` statements are possible.
For more complex conditions, splitting them across multiple `FILTER`
statements can improve readability.
```

```
[NOTE]
====
Performance is usually better with a single `WHERE`, since multiple
`FILTER` statements are currently not optimized but executed linearly one
after another.
====

[source,cypher]
```

MATCH (m:manual)-[:describes]->(p:product) FILTER p.weight > 1000 FILTER m.language = 'en' RETURN m.title,p.id

```
This query is equivalent to

[source,cypher]
```

MATCH (m:manual)-[:describes]->(p:product) WHERE p.weight > 1000 AND m.language = 'en' RETURN m.title,p.id

```
=== Combining with MATCH

In a linear query, multiple `MATCH` sub-queries can be formulated, each
with an optional `WHERE` condition:

[source,cypher]
```

MATCH (p:product) WHERE p.weight > 1000 MATCH (m:manual)-[:describes]->(p) WHERE m.language = 'en' RETURN m.title,p.id

```
This query is equivalent to

[source,cypher]
```

MATCH (m:manual)-[:describes]->(p:product) WHERE p.weight > 1000 AND m.language = 'en' RETURN m.title,p.id

```
For more complex queries, splitting them across multiple sub-queries can
improve readability.

With multiple sub-queries, each sub-query except the first merges its
result with the result of the previous sub-query.
```

```
The optional following `WHERE` condition is applied to the merged result.
You can therefore also reference variables from previous sub-queries.

The results are merged based on the shared fields of the sub-queries.
In the example above, that is the variable `p`.
If there are no shared fields, a Cartesian product is formed.

[NOTE]
====
Performance is usually better with a merged query than with a query split
across multiple `MATCH` conditions.

Multiple `MATCH` conditions without shared fields should be avoided at all
costs, as a Cartesian product requires significant memory and time.
====

=== OPTIONAL MATCH

A special form of MATCH is `OPTIONAL MATCH`.
This allows you to add optional properties to the result.

If the search conditions do not match any elements of the graph, the
result of the sub-query is a table with one row of `NULL` values instead
of an empty table.
When merging with the previous sub-query, this causes the results of the
previous sub-query to be preserved.

[NOTE]
====
Only properties are added, no new rows.
For unioning sub-queries, use <<gql-composition-table-union>>
====

Example:

[source,cypher]
```

MATCH (p:product) WHERE p.weight > 10000 OPTIONAL MATCH (p:product) WHERE p.tag = 'heavy' RETURN p.name as product, p.weight as weight, p.tag as tag

```
The result also contains rows in which the tag field is empty:

[%header,cols="a,a,a"]
|===
|product
```

```
|weight
|tag

|Anvil A1
|17200
|heavy

|PowerMole 3000
|27150
|

|===

Without `OPTIONAL` the result only contains rows in which the `tag` field
has the value `heavy`:

[source,cypher]
```

MATCH (p:product) WHERE p.weight > 10000 MATCH (p:product) WHERE p.tag = 'heavy' RETURN p.name as product, p.weight as weight, p.tag as tag

```
[%header,cols="a,a,a"]
|===
|product
|weight
|tag

|Anvil A1
|17200
|heavy

|===


== Composite Queries

A linear query returns a table with the fields defined in `RETURN` as its
result.
The results of two linear queries can be combined into one table using
various operations.

The basic structure of a composite query is `<Linear Query 1> <OPERATOR>
<Linear Query 2>`.
The operators always combine exactly two tables.
If you specify multiple operators, they are executed individually in
```

```
sequence.

=== Set Operators

With set operators, the individual rows of the two tables are combined
using set operations.

The tables must have the same fields for this.
The order of the fields is not relevant.
If the fields differ, the query results in an error message.

Duplicates can be removed with the keyword `DISTINCT` after the operator
(see <<gql-composition-distinct-example, example>> at UNION)

[#gql-composition-table-union]
==== UNION

With `UNION`, the rows of both tables are added to the result table.

[source,cypher]
```

MATCH (p:product) WHERE p.weight > 10000 RETURN p.name as product, p.weight as weight, p.tag as tag UNION MATCH (p:product) WHERE p.tag = 'heavy' RETURN p.name as product, p.weight as weight, p.tag as tag

```
[%header,cols="a,a,a"]
|===
|product
|weight
|tag

|Anvil A1
|17200
|heavy

|Anvil A1
|17200
|heavy

|PowerMole 3000
|27150
|

|===
```

```
[#gql-composition-distinct-example]
Due to the union, the result contains duplicates.
These can be removed with `UNION DISTINCT`.

[source,cypher]
```

MATCH (p:product) WHERE p.weight > 10000 RETURN p.name as product, p.weight as weight, p.tag as tag UNION DISTINCT MATCH (p:product) WHERE p.tag = 'heavy' RETURN p.name as product, p.weight as weight, p.tag as tag

```
[%header,cols="a,a,a"]
|===
|product
|weight
|tag

|Anvil A1
|17200
|heavy

|PowerMole 3000
|27150
|

|===



=== INTERSECT

With `INTERSECT`, rows that are present in both result tables are added to
the result table.

=== EXCEPT

With `EXCEPT`, rows from the first table that are not contained in the
second table are added.

[NOTE]
====
`INTERSECT` and `EXCEPT` do not produce duplicates, but `DISTINCT` can
still be used, for example if the first table contains duplicates.
====

== OTHERWISE
```

```
`<Linear Query 1> OTHERWISE <Linear Query 2>` returns the result of the
first query if it is not empty.
The second query is not executed.

If the result is empty, the result of the second query is returned
instead.

In contrast to `UNION`, `INTERSECT`, and `EXCEPT`, the two tables do not
need to have the same fields.

[source,cypher]
```

MATCH (p:product)-[:producedBy]→(m:company) WHERE m.name = $manufacturer OR m.id = $manufacturer RETURN p.name as product, m.id as manufacturer-id, p.producer as producer-tag OTHERWISE MATCH (p:product) WHERE p.producer = $manufacturer RETURN p.name as product, p.producer as producer-tag

```
Query with `$manufacturer` = "M173621"

[%header,cols="a,a,a"]
|===
|product
|manufacturer-id
|producer-tag

|Anvil A1
|M173621
|heavy-industries

|PowerMole 3000
|M173621
|heavy-corp

|===

Query with `$manufacturer` = "heavy-corp"

[%header,cols="a,a"]
|===
|product
|producer-tag

|PowerMole 3000
|heavy-corp
```

```
|===

[#gql-composition-next]
== NEXT

Multiple linear queries can be chained together with `NEXT`.
The partial result tables are joined by a Cartesian product.
You can reference variables from preceding sub-queries.

[source,cypher]
```

MATCH (p:product) WHERE p.rating > 4 RETURN p NEXT MATCH (p) where p.price < 1000 RETURN p.name, p.rating, p.price

```
It is also possible to access aggregated values, which is not possible in
the `WHERE` part.

[source,cypher]
```

MATCH (c:company)-[:produces]→(p:product) RETURN c.name as company, COUNT(p) as products GROUP BY c NEXT FILTER products >= 2 RETURN company, products

```
[#gql-composition-call]
== CALL

With `CALL`, other queries are called from within a GQL query.
GQL distinguishes between inline and named procedures.
Procedures are evaluated row by row based on the current intermediate
result.
If the called procedure returns more than one result, one row is inserted
into the caller's result table for each result row.
Conversely, this also means that an empty result from the called procedure
removes the underlying row from the caller's result table.

[NOTE]
====
All procedures are evaluated row by row based on the current intermediate
result.
This can result in significant performance degradation.
It is therefore often better to avoid a procedure call and integrate the
statements into a preceding `MATCH` or `FILTER`.
====

[#gql-composition-inline-procedures]
```

```
=== Inline Procedures

Inline procedures are encapsulated GQL queries that are integrated
directly within another query.
This enables a logical structuring of the query.
You can control which variables are passed to the inline procedure and
which return values are included in the result table of the overall query.
Variables that are not passed to a procedure are not visible to it and can
also be redefined within the procedure.

[example]
====
The following example uses an inline procedure to determine the owner of
each company `c` from the preceding `MATCH`.
The variable `p` is not passed to the procedure and can therefore be
reassigned within the procedure for persons.
When returning the persons from the procedure, an alias must be assigned,
since `p` in the outer context refers to products.

[source,cypher]
```

MATCH (c:company)-[:produces]→(p:product) CALL (c) { MATCH (c)-[:ownedBy]→(p:person) RETURN p AS owner } RETURN c AS company, owner, COLLECT_LIST(p) AS products GROUP BY c

```
====


The list of passed variables (`(c)`) is optional.
If it is not specified, all variables from the outer context are passed
implicitly.
An empty variable list `()` provides no existing variables to the
procedure.

[NOTE]
====
The GQL specification prohibits inline procedures from returning variables
that already exist in the outer context.
In the example above, neither `RETURN p` nor `RETURN *` would therefore be
allowed, since the variable `p` is used for products in the outer context.
====

=== Calling External Queries

By calling named procedures, other queries stored in the knowledge graph
can be integrated into a GQL query.
These can be either further GQL queries or other queries, such as
```

```
structured queries or full-text searches.
The keyword `CALL` is also used for named procedures.
This requires the registry key of the query to be called.
With the keyword `YIELD` you specify which columns of the called procedure
should be available in the further query.

[example]
====
Assuming the query from the example <<gql-composition-inline-procedures>>
is registered under the registry key `productsByCompany`, it can be
integrated into a new query via the following `CALL`:

[source,cypher]
```

CALL productsByCompany() YIELD company, products RETURN company, SIZE(products) AS productCount

```
====

[NOTE]
====
The GQL specification assumes that the types and names of the fields
returned by a called procedure are statically known.
The syntax therefore also allows calls without `YIELD`, implicitly
adopting all fields of the called procedure.
This is currently not possible in i-views, since the return value of a
called GQL procedure is only known at runtime.
====

==== Parameterization

If the called procedure defines parameters, these can be passed in the
form of a record.
Within the parameter record, each key must correspond to the name of a
parameter.
The following examples use `FOR ... IN` to iterate over a list of values
-- see <<gql-further-statements-for,Iterating with FOR>>.

[example]
====
Query with registry key `productsByCompanyWithName` and parameter
`$companyName`:

[source,cypher]
```

MATCH (c:company)-[:produces]→(p:product) WHERE c.name = $companyName RETURN p AS product

```
Call as a named procedure with parameter `$companyName`:

[source,cypher]
```

FOR name IN ['ACME', 'Mustermann GmbH'] CALL productsByCompanyWithName({companyName: name}) YIELD product RETURN product, name AS `manufactured by`

```
The result is a table of products from both companies.
====

If the called procedure defines parameters that are not explicitly passed
in the parameter record during the call, the parameter value of the same
name from the calling GQL query is used instead.
If the parameter is not set for the calling query, a runtime error occurs.

[example]
====
The called query `productsByCompanyWithName` defines the parameter
`$companyName`.
Since no parameters are set during the `CALL`, the value is taken from the
`$companyName` parameter of the caller.

[source,cypher]
```

CALL productsByCompanyWithName() YIELD product RETURN product, $companyName AS manufacturedBy

```
====

==== Calling Non-GQL Queries

If the called procedure is not a GQL query but, for example, a structured
query, the returned table contains exactly one column with the name
`_result`.
This column contains the result objects, which have the type `NODE` in
GQL.

[NOTE]
====
Calling queries that produce hits not based on knowledge graph objects is
```

```
currently not possible.
In this case, the call returns an empty result table.
====

Parameterization works identically to calling GQL queries using a
parameter record.
Some query types, for example full-text searches, require a parameter
named `searchString` containing the string to search for.

[example]
====
In the following example, a full-text search for newspaper articles
containing one of the two company names is performed.
The return value is a table with the name of the company and the name and
date of each found newspaper article.

[source,cypher]
```

FOR name IN ['ACME', 'Mustermann GmbH'] CALL pressReleaseFulltextSearch({searchString: name}) YIELD _result AS pressRelease RETURN name, pressRelease.name, pressRelease.releaseDate

```
====


=== OPTIONAL CALL
If a called procedure returns an empty result table, the entire row for
which the procedure was called is removed from the overall result by
default.
If this is not desired, you can call `CALL` with the `OPTIONAL` prefix.
All fields of the `YIELD` expression are then set to `NULL` if the called
procedure returns an empty result for the row.

[example]
====
If "ACME" is not mentioned in any newspaper article, the result table of
the following GQL program still contains a row with `name` "ACME" in which
`release` and `date` are set to `NULL`.

[source,cypher]
```

FOR name IN ['ACME', 'Mustermann GmbH'] OPTIONAL CALL pressReleaseFulltextSearch({searchString: name}) YIELD _result AS pressRelease RETURN name, pressRelease.name AS release, pressRelease.releaseDate AS date

```
:leveloffset: 4
```

```
:leveloffset: 4

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#gql-further-statements]
= Further Statements

This section describes statements that do not belong to any of the other
chapters:
<<gql-further-statements-let-value,Variable Assignment with `LET` and
`VALUE`>> and <<gql-further-statements-for,Iterating with `FOR`>>.

[#gql-further-statements-let-value]
== Variable Assignment with LET and VALUE

=== LET

With `LET` you can bind expressions to variables in order to reuse them
```

```
multiple times in the rest of the query.
Multiple assignments can be chained with a comma.

[source,cypher]
```

MATCH (p:product) LET discounted = p.price * 0.9, label = UPPER(p.name) WHERE discounted < 100 RETURN label, p.price, discounted ORDER BY discounted

```
If the binding should be restricted to a single expression, you can use
the `LET ... IN ... END` form.
The variables are then only visible in the expression between `IN` and
`END`.

[source,cypher]
```

MATCH (p:product) RETURN p.name, LET discounted = p.price * 0.9 IN CASE WHEN discounted < 100 THEN discounted ELSE p.price END END

```
=== VALUE

`VALUE` is an alternative form of variable assignment that can only be
used at the beginning of a procedure body.
In contrast to `LET`, only one variable can be defined per `VALUE`
statement; however, multiple consecutive `VALUE` statements are allowed.

[source,cypher]
```

VALUE minRating = 4 VALUE category = 'electronics' MATCH (p:product) WHERE p.rating >= minRating AND p.category = category RETURN p.name, p.rating

```
[#gql-further-statements-for]
== Iterating with FOR

With `FOR` you can iterate over the elements of a list.
For each element of the list, one row is inserted into the intermediate
result.

[source,cypher]
```

FOR name IN ['ACME', 'Mustermann GmbH'] RETURN name

```
[%header,cols="a"]
|====
|name

|ACME

|Mustermann GmbH

|====

`FOR` can be combined with subsequent statements that reference the
iteration variable.

[source,cypher]
```

FOR name IN ['ACME', 'Mustermann GmbH'] MATCH (c:company) WHERE c.name = name RETURN c.name, c.foundingYear

```
The list can also be a computed value, for example the result of
`COLLECT_LIST` in a preceding sub-query:

[source,cypher]
```

MATCH (c:company)-[:produces]→(p:product) RETURN COLLECT_LIST(p.name) AS productNames GROUP BY c NEXT FOR name IN productNames RETURN name

```
:leveloffset: 4
:leveloffset: 4

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
```

```
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#gql-parameters]
= Query Parameters

GQL queries can define parameters whose values are passed at runtime.
A parameter is identified in the query text by a leading `$`.

[source,cypher]
```

MATCH (p:product) WHERE p.rating > $minRating AND p.category = $category RETURN p.name, p.rating

```
Parameters can be used anywhere a literal value is allowed: in `WHERE`
conditions, `RETURN` expressions, `ORDER BY`, and as arguments to
functions.

[NOTE]
====
If a parameter is used in a query but no value is passed for it at
runtime, this results in an error.
There is no implicit deactivation of conditions for missing parameters.
====

== Passing Parameter Values

How parameters are passed depends on the execution environment:

REST API::
Parameters are passed as request parameters (GET, POST with text body) or
in the JSON object `queryParameters` (POST with form data).
```

```
See <<gql-rest-api,GQL REST API>>.

JavaScript API::
Parameters are passed as an object to `perform()`, where the property
names correspond to the parameter names.
See <<gql-js-api,GQL JavaScript API>>.

Workbench::
Parameters can be entered directly in the Workbench.

== Parameter Value Types

Parameter values are passed as strings and implicitly converted to the
type appropriate for the context.
With the JavaScript API, values can also be passed as native JavaScript
types.

[NOTE]
====
Lists as parameter values are not transparent: if an array value is
passed, it must be explicitly evaluated as a list in the query, for
example with `FOR ... IN $myListParam`.
====

:leveloffset: 4
:leveloffset: 4

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
```

```
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#gql-iviews-specifics]
= i-views-specific Language Extensions

== Mapping i-views Attribute Types to GQL Value Types

Not all i-views attribute types have a direct equivalent in the GQL
standard.
The following table shows how i-views attribute types are mapped to GQL
value types.

[%header,cols="2a,2a,3a"]
|===
|i-views attribute type
|GQL value type
|Note

|String
|String
|Multilingual values: see <<gql-iviews-specifics-multilingual-attributes>>

|Integer
|Integer
|

|Float
|Float
|Double precision only

|Boolean
|`TRUE` / `FALSE`
|

|Date
|`DATE`
|
```

```
|Time
|`TIME`
|

|Date and time
|`DATETIME` / `TIMESTAMP`
|

|Flexible time
|`DATE`, `TIME` or `DATETIME`
|Depending on stored value

|Internet shortcut
|String
|Multilingual values: see <<gql-iviews-specifics-multilingual-attributes>>

|Selection
|String
|

|File
|String
|File name is returned as a string

|Interval
|--
|Can only be output as a string

|Geometry / Geographical position
|--
|Can only be output as a string (WKT format)

|Color value
|--
|Can only be output as a string

|===

== Extended Value Comparisons

i-views offers more comparison operators than are found in the GQL
standard.
For example, i-views allows comparisons with regular expressions,
wildcards, and case-insensitive matching, as well as comparison operators
for special value types such as intervals or geometries.
```

```
Using the `IV_COMPARE` function, you can access all native operators of i-
views.
The function expects two arguments to compare, as well as the name of a
comparison operator as a string.
`IV_COMPARE` returns either `TRUE` or `FALSE`.
The function can also be used directly in combination with a `WHERE`
clause without an additional comparison with `TRUE` or `FALSE`.

[example]
====
[source,cypher]
```

MATCH (c:city) WHERE IV_COMPARE(c.name, '**b**', 'equal') RETURN c.name

```
[%header,cols="1a"]
|===
|c.name

|Berlin
|Hamburg
|...

|===

[source,cypher]
```

MATCH (c:city) WHERE IV_COMPARE(c.name, '(Bad)?H(a|o)mburg', 'equalRegex') RETURN c.name

```
[%header,cols="1a"]
|===
|c.name

|Hamburg
|Bad Homburg
|...

|===
====

You can find the complete list of operators with their respective
identifiers in the technical handbook.

Optional additional parameters can be passed as a record in the fourth
```

```
argument.
The available parameters depend on the respective operator.

[%header,cols="1a,1a,1a,1a"]
|===
|Operator
|Parameter
|Default
|Description

|String comparison operators
|`allowWildcards`
|`TRUE`
|`FALSE` disables `*` and `?` as wildcards for arbitrary characters

|
|`ignoreCase`
|`TRUE`
|`FALSE` enables case-sensitive matching

|Distance comparison operators
|`distanceValue`
|`0`
|Maximum distance allowed between the two values

|===

[example]
====
In this example, cities are found whose geodesic distance to Darmstadt is
at most 100 kilometres.
There is currently no GQL equivalent for the geometry value type, so it
would not be possible to perform any comparison with such values using
standard operators.
`IV_COMPARE` also supports these value types and automatically converts
the string representation of the comparison value.

[source,cypher]
```

MATCH (c:city) WHERE IV_COMPARE(c.area, 'SRID=4326;POINT(8.6512 49.8728)', 'geodesicDistance', { distanceValue: '100km' }) RETURN c.name

```

====

[NOTE]
```

```
====
At least one of the values being compared in `IV_COMPARE` must be a
reference to an attribute.
It is not possible to compare two literal values with `IV_COMPARE`.
====

== Multi-valued Attributes

In general, attributes of objects in the knowledge graph are mapped to
properties in GQL.
Following the GQL standard, each property has either exactly one material
value or it is `NULL`.
i-views, however, supports multi-valued attributes.
Although GQL supports list types for properties, multi-valued attributes
are deliberately not mapped to lists, because working with lists in GQL is
comparatively complex.
Multi-valued attributes have no inherent ordering of attribute values, for
example, so a comparison with a sorted list would generally not be
possible.
Instead, multi-valued attributes are split across a dynamic number of rows
in the query result, analogously to multi-valued relations (edges).

[example]
====
[source,cypher]
```

MATCH (p:person) RETURN p.name, p.nickname, UPPER(p.nickname) AS uppercase

```
[%header,cols="1a,1a,1a"]
|===
|p.name
|p.nickname
|uppercase

|"Jonathan"
|"Jon"
|"JON"

|"Jonathan"
|"Nathan"
|"NATHAN"

|"Jonathan"
|"Nat"
|"NAT"
```

```
|===
====

In each row, all references to the multi-valued attribute refer to the
same value.
This means in particular that when the attribute is referenced in a
`WHERE` clause, rows for non-matching attribute values are also removed.

[example]
====
[source,cypher]
```

MATCH (p:person) WHERE p.nickname = "Nathan" RETURN p.name, p.nickname, UPPER(p.nickname) AS uppercase

```
[%header,cols="1a,1a,1a"]
|===
|p.name
|p.nickname
|uppercase

|"Jonathan"
|"Nathan"
|"NATHAN"

|===
====

//TODO: How to output all nicknames of persons where one nickname is
Nathan
//Task 39649

== Multilingual Attribute Values

In general, attributes of objects in the knowledge graph are mapped to
properties in GQL.
Following the GQL standard, each property has either exactly one material
value or it is `NULL`.
i-views, however, supports multilingual attribute values for string, URL,
and file attributes.
To remain as standards-compatible as possible, querying a multilingual
attribute returns the value that best matches the current language of the
application.
This may be the system language, for example, or in the context of a web
```

```
application the language preferred by the browser.
This applies both to queries in `WHERE` clauses and to return values in
`RETURN` statements.

[example]
====
The following query only returns results in a German application context,
because `c.name` returns the value matching the application language by
default.
The return value gives the name in the application language.
[source,cypher]
```

MATCH (c:city) WHERE c.name = 'Köln' RETURN c.name

```
====

To access specific translations of an attribute value, there are two i-
views-specific language extensions.

Language access via qualifier::
If the desired language is statically known, use a qualifier on the
property reference.
It is introduced by a hash sign (`\#`) followed by an ISO 639-1 (`de`),
ISO 639-2 (`ger`), or locale identifier (`de_DE`).
If an identifier is queried for which no translated attribute value
exists, `NULL` is returned.
+
[example]
====
The following query matches the German city name regardless of the
application language.
The return value gives the name in the application language, in German,
and in US English (provided a corresponding translation exists).
[source,cypher]
```

MATCH (c:city) WHERE c.name\#de = 'Köln' RETURN c.name, c.name\#de, c.name\#en_US

```
====

Language access with `IV_VALUE`::
The `IV_VALUE` function supports an optional second argument specifying a
language as a string.
Here too, an ISO 639-1 (`'de'`), ISO 639-2 (`'ger'`), or locale identifier
(`'de_DE'`) is expected.
```

```
This variant is slightly less compact, but also allows the use of
variables or property references for dynamically returning translations.
+
[example]
====
The following query matches the German city name regardless of the
application language.
The return value gives three rows: the German, the US English, and the
native-language value.
[source,cypher]
```

MATCH (c:city) WHERE IV_VALUE(c.name, 'de') = 'Köln' FOR lang IN ['de', 'en_US', city.nativeLanguageID] RETURN IV_VALUE(c.name, lang)

```
====
+
Using the `IV_VALUE` function, it is also possible to query the translated
value from an attribute node.
+
[example]
====
The following query returns all German values of the name attribute.
[source,cypher]
```

MATCH (nameAttribute:name) RETURN IV_VALUE(nameAttribute, 'de')

```
====

== Accessing Meta-properties

The GQL graph model defines nodes and edges, each of which can have
properties.
Edges always connect exactly two nodes.
Neither edges nor properties can themselves be the source of an edge or
property.
The i-views graph model allows such meta-properties.
All elements of the knowledge graph (objects, relations, attributes, and
types) have a schema that allows the definition of attributes and
relations.

In this GQL implementation, attributes and relations can therefore also be
treated as nodes.
They are thus first-class elements, just like objects.
```

```
[example]
====
The name attribute with the internal name `name` can be found as a node
just like any other object.
[source,cypher]
```

MATCH (n:name) RETURN n

```
====

In this way it is possible to access meta-properties using normal GQL
syntax.

[example]
====
Accessing the meta-attribute `date` of the relation `editedBy`:
[source,cypher]
```

MATCH (r:editedBy) RETURN r.date

```
Accessing the meta-relation `signedOffBy` of the relation `editedBy` and
the meta-meta-attribute `date`:
[source,cypher]
```

MATCH (r1:editedBy)-[r2:signedOffBy]→(p:person) RETURN p.name, r2.date

```
====

In combination with the rest of the GQL syntax, it is sometimes necessary
to reinterpret an edge as a node within a query.
This is made possible by the `\#node` qualifier:

[example]
====
In this example, `r` is first used as the identifier of an edge whose
target is to be part of the output.
To additionally access a meta-relation of `r`, `r` must be referenced as a
node using `r\#node`.
[source,cypher]
```

MATCH (doc:document)-[r:editedBy]→(editor:person), (r\#node)-[:signedOffBy]→(signer:person) RETURN doc.name, editor.name, signer.name

```
====

== Accessing System Relations

System relations can be used as edges in a query just like ordinary
relations.
Each system relation has a predefined label with the prefix `iv_`, by
which it can be referenced (`-[r:iv_hasTarget]->`).

[%header,cols="1a,1a"]
|===
|System relation
|Label

|Defined for
|`iv_hasDomain`

|Defined for instances of
|`iv_domainOfInstancesOf`

|Defined for subtypes of
|`iv_domainOfSubtypesOf`

|Domain of
|`iv_isDomainOf`

|Supplements object
|`iv_hasSupplementalInstance`

|Supplements instances of
|`iv_isSupplementalInstanceOf`

|Extends object
|`iv_extends`

|Extends instances of
|`iv_extendsInstancesOf`

|Has extension
|`iv_extendedBy`

|Has instance
|`iv_hasInstance`

|Has target
```

```
|`iv_hasTarget`

|Inverse relation
|`iv_hasInverseRelation`

|Is supertype of
|`iv_hasSubType`

|Is instance of
|`iv_hasType`

|Is subtype of
|`iv_hasSuperType`

|Is target of
|`iv_isTargetOf`

|Instances can be extended by
|`iv_instancesExtendedBy`

|Instances are domain of
|`iv_instanceIsDomainOf`

|Types are domain of
|`iv_typeIsDomainOf`

|===

== Access Parameters

Depending on the execution environment, certain access parameters are
available, such as the current user, the accessed element, or the
application configuration.
These parameters can be queried using the `IV_ACCESS_PARAMETER` function.
The function requires a string as an argument specifying the parameter
name.
Since all access parameters reference objects in the knowledge graph, the
result of the `IV_ACCESS_PARAMETER` function is always a node.

[example]
====
Outputting all access parameters:
[source,cypher]
```

FOR accessParam IN ['user', 'accessedObject', 'applicationTopic', 'property', 'topic', 'inverseTopic',

'inverseRelation', 'inverseRelationConcept', 'core', 'primaryProperty', 'primaryTopic', 'primaryCoreTopic', 'inversePrimaryCoreTopic'] RETURN accessParam, IV_ACCESS_PARAMETER(accessParam)

```
All persons known by the current user:
[source,cypher]
```

MATCH (u:person)-[:knows]→(p:person) WHERE u = IV_ACCESS_PARAMETER("user") RETURN p.name

```
:leveloffset: 4
:leveloffset: 4

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

:search-example: Example
```

```
:structured-query: Structured query
:gql: GQL

[#gql-structured-queries]
= GQL Queries Equivalent to Structured Queries

The following section shows some structured queries and equivalent GQL
queries.

[NOTE]
An automatic translation of structured queries into GQL (or vice versa) is
not planned.

== Properties

Object has a relation::

{search-example}:::
Persons who own a book

{structured-query}:::
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/searches-
queries/gql/gql-structured-queries/gfx/relation.png[]

{gql}:::
+
[source,cypher]
```

MATCH (p:person)-[:owns]→(b:book) RETURN p.name, b.name

```

+
See <<gql-patterns-edges,Edges>>

Number of relations (cardinality)::

{search-example}:::
Persons who own at least three books

{structured-query}:::
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/searches-
queries/gql/gql-structured-queries/gfx/relation-with-cardinality.png[]

{gql}:::
```

```
+
[source,cypher]
```

MATCH (p:person)-[o:owns]→(b:book) RETURN p.name as name, COUNT(o) as owns GROUP BY p NEXT FILTER owns >= 3 RETURN name, owns

```
+
Cardinality is achieved through aggregation (`GROUP BY`) and counting
(`COUNT`).
Since aggregation is only possible in `RETURN` statements, you cannot
count in the `WHERE` part.
Since you cannot filter in `RETURN`, a `FILTER` linked by <<gql-
composition-next,`NEXT`>> is appended.

This way you also get the actual count in the result.

Condition on attribute value::

{search-example}:::
Persons born on or after 1 January 2000

{structured-query}:::
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/searches-
queries/gql/gql-structured-queries/gfx/attribute-greater-than.png[]

{gql}:::
+
[source,cypher]
```

MATCH (p:person) WHERE p.dateOfBirth >= DATE "2000-01-01" RETURN p.name, p.dateOfBirth

```
Comparing attribute values::

{search-example}:::
Persons who know a person who is older than themselves

{structured-query}:::
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/searches-
queries/gql/gql-structured-queries/gfx/attribute-reference.png[]

{gql}:::
+
```

```
[source,cypher]
```

MATCH (p:person)-[:knows]→(q:person) WHERE p.dateOfBirth > q.dateOfBirth RETURN p.name, p.dateOfBirth, q.name, q.dateOfBirth

```
== Types

Subtypes::

{search-example}:::
Type "Product" and all subtypes

{structured-query}:::
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/searches-
queries/gql/gql-structured-queries/gfx/subtypes.png[]

{gql}:::
+
[source,cypher]
```

MATCH (p:product\#type) RETURN p.name

```
+
The suffix `\#type` on the label `product` indicates that subtypes rather
than objects are to be searched for.

Objects of types without inheritance::

{search-example}:::
Objects of type "Product", excluding objects of subtypes of "Product"

{structured-query}:::
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/searches-
queries/gql/gql-structured-queries/gfx/no-inheritance.png[]

{gql}:::
+
[source,cypher]
```

MATCH (p:product\#exact) RETURN p.name

```
+
The inheritance that is active by default is disabled by the suffix
`\#exact` on the label `product`.

Objects of multiple types::

{search-example}:::
Objects of type "Product" or "Company"

{structured-query}:::
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/searches-
queries/gql/gql-structured-queries/gfx/multi-type.png[]

{gql}:::
+
[source,cypher]
```

MATCH (t:product|company) RETURN t.name

```
+
You can specify multiple alternative labels using `|`.
+
See <<gql-patterns-labels,Labels>>

== Identification

[NOTE]
Do not reference internal IDs in GQL queries, as these are not preserved
when objects are transferred to other graphs.
Use an identifying attribute instead.

Specifying an element::

{search-example}:::
Persons who own the product _Power Mole 3000_

{structured-query}:::
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/searches-
queries/gql/gql-structured-queries/gfx/fixed-topic.png[]

{gql}:::
+
```

```
[source,cypher]
```

MATCH (p:person)-[:owns]→(o:product) WHERE o."RDF-about" = 'http://example.org/#powerMole3000' RETURN p.name, o.name

```
+
The target object is specified here by an identifying attribute.

Identifying an element by ID::

{search-example}:::
Persons who own a product specified by an ID parameter

{structured-query}:::
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/searches-
queries/gql/gql-structured-queries/gfx/frame-id.png[]

{gql}:::
+
[source,cypher]
```

MATCH (p:person)-[:owns]→(o:product) WHERE ELEMENT_ID(o) = $id RETURN p.name, o.name

```
The element ID is passed to the GQL query via the `id` parameter.

== Structure

Alternatives::

{search-example}:::
Products that contain other products or are contained in other products

{structured-query}:::
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/searches-
queries/gql/gql-structured-queries/gfx/alternative.png[]

{gql}:::
+
[source,cypher]
```

MATCH (p:product)-[:contains]→(o:product) RETURN p.name as name, o.name as related UNION

MATCH (p:product)-[:containedIn]→(o:product) RETURN p.name as name, o.name as related

```
+
In general, you can implement alternatives as <<gql-composition-table-union,`UNION`>>.
+
If the alternatives only contain attribute conditions, you can also use `OR` in the `WHERE` part, as the following example shows.

Alternatives (simple)::

{search-example}:::
Products that have a rating of >= 4 or no rating

{structured-query}:::
image:/builds/i-views/documentation/documentation-asciidoc/build/pdf/users-manual-en/knowledge-builder/searches-queries/gql/gql-structured-queries/gfx/alternative-with-attributes.png[]

{gql}:::
+
[source,cypher]
```

MATCH (p:product) WHERE p.rating >= 4 OR p.rating IS NULL RETURN p.name as name, p.rating as rating ORDER BY rating DESC

```
+
Instead of `UNION`, an `OR` condition is used (see <<gql-expressions-logical-operators,logical operators>>).
This is possible when only attributes and other expressions are needed in the condition.
For alternatives involving relations, `UNION` must be used.

References::

{search-example}:::
Persons who know two different persons who own the same product

{structured-query}:::
image:/builds/i-views/documentation/documentation-asciidoc/build/pdf/users-manual-en/knowledge-builder/searches-queries/gql/gql-structured-queries/gfx/reference.png[]

{gql}:::
```

```
+
[source,cypher]
```

MATCH (p:person)-[:knows]→(q1)-[:owns]→(t), (p)-[:knows]→(q2)-[:owns]→(t) WHERE q1 <> q2 RETURN p.name, q1.name, q2.name, t.name

```
+
Identity can be expressed by using the same variable `t` in both
conditions.

:leveloffset: 4
:leveloffset: 4

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#gql-rest-api]
= GQL REST API
```

```
Using the REST API you can execute GQL queries and output the result in
various formats.

To set up the API, create a new resource of type _Built-In Resource_ in a
REST service and specify one of the predefined GQL IDs as the _REST
resource id_:

`GQLQueryResource`::
Allows the execution of registered queries and of queries specified in the
request (ad-hoc queries).

`GQLRegisteredQueryResource`::
Only registered queries can be executed.
Requests with an ad-hoc query result in an error.

`GQLQueryValidationResource`::
Validates the query but does not execute it.
For valid queries, the query and query parameters are returned in JSON
format as the result.
For errors, a `BAD REQUEST` with an error message in JSON format is
returned.

The resource supports `GET` and `POST` requests.

== GET

Only registered queries can be called.

The registration key of the query is specified as the parameter `queryId`.
All parameters other than `queryId`, `query`, and `queryParameters` are
evaluated as query parameters.

You can define the parameters as query, path, or header parameters when
configuring the REST method.
Without configuration, a query parameter is expected.

[NOTE]
The parameters `query` and `queryParameters` are not supported for GET
requests.

== POST with Form Data

For a POST request with Content-Type `application/x-www-form-urlencoded`
or `multipart/form-data`, the following form parameters can be specified:
```

```
`queryId`::
The registration key of a registered query

`query`::
The source text of a GQL query.
For the resource `GQLRegisteredQueryResource`, this form parameter results
in an error, as ad-hoc queries are not permitted.

`queryParameters`::
Query parameters in JSON syntax

[NOTE]
The parameters `query` and `queryParameters` cannot be specified at the
same time.

== POST without Form Data

If you use a content type other than the two form types, the body of the
POST request is expected to contain the source text of a GQL query.

[NOTE]
Since there is currently no officially registered content type for GQL,
specify `text/plain` with a charset.

The query parameters are expected as request parameters, analogously to
`GET`.

For the resource `GQLRegisteredQueryResource`, the request results in an
error, as ad-hoc queries are not permitted.

== Output

The desired output format is selected via the `Accept` header field of the
request.

`text/csv`::
Output as a CSV file.
The table contains one column per field specified in `RETURN`, with the
same name.
- Encoding `UTF-8`
- Separator `;`
- Values are enclosed in quotation marks.
- First row contains headers.

`application/vnd.openxmlformats-officedocument.spreadsheetml.sheet`::
Output as an Excel file.
```

```
`application/json`::
Output as a JSON file.
The file contains an array of objects.
Each object corresponds to one search result and contains the fields
specified in `RETURN`.
+
[NOTE]
====
If a field name contains a dot, the property keys also contain a dot.
While this is permitted in JSON, it may be inconvenient for further
processing.
Use `AS` to choose a name without a dot, for example `MATCH (p:person)
RETURN p.name AS personName`.
====

`text/plain`::
Textual representation of a table.
This format is suitable for testing.
For machine processing, use a different format.

== Examples

The following examples address a REST service `gql` with a built-in
resource `GQLQueryResource` registered under the path `search`.

GET::
Calling the registered query `persons` with query parameter `name`
+
[source,shell]
```

GET /gql/search?queryId=persons&name=Goethe Host: localhost:8815 Accept: application/json

```
+
With curl:
+
[source,shell]
```

curl -H "Accept: application/json" "http://localhost:8815/gql/search?queryId=persons&name=Goethe"

```
POST with form data::
Calling an ad-hoc query:
+
```

```
[source,shell]
```

curl -H "Accept: text/csv" -F query="MATCH (p:person) RETURN p.name, p.birthday" "http://localhost:8815/gql/search"

```
+
Calling the registered query `persons` with query parameter `name` as a
form parameter:
+
[source,shell]
```

curl -H "Accept: text/plain" -F queryId="persons" -F name="Fleißer" "http://localhost:8815/gql/search"

```
+
Calling an ad-hoc query with query parameters in JSON format:
+
[source,shell]
```

curl -H "Accept: text/plain" -F query="MATCH (p:product) WHERE p.rating > $rating RETURN p.name, p.rating" -F "queryParameters={\"rating\": 3}" "http://localhost:8815/gql/search"

```
POST with text body::
Calling an ad-hoc query with query parameters as URL query parameters:
+
[source,shell]
```

curl -H "Accept: text/plain" -H "Content-type: text/plain; charset=utf-8" -d "MATCH (p:product) WHERE p.rating > $rating RETURN p.name, p.rating" "http://localhost:8815/gql/search?rating=4"

```
:leveloffset: 4
:leveloffset: 4

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
```

```
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#gql-js-api]
= GQL JavaScript API

Using the JavaScript API you can execute GQL queries programmatically and
access the result table.

== Creating a GQL Query

A GQL query is created as an instance of `$k.GQLQuery`.
The constructor optionally accepts the query text as a string.

[source,js]
```

const query = new $k.GQLQuery('MATCH (p:product) RETURN p.name, p.price')

```
Alternatively, you can set the query text afterwards with `setSource()` or
retrieve it with `source()`.

[source,js]
```

const query = new $k.GQLQuery() query.setSource('MATCH (p:product) RETURN p.name, p.price')

```
To execute a registered query as a GQL query, call `$k.Registry.query()`
with the registration key.
The result is also a `$k.GQLQuery` object.

[source,js]
```

const query = $k.Registry.query('products')

```
Note that calling `setSource()` on a registered GQL query modifies and
saves the underlying source text.
This requires a write transaction.

== Setting Parameters

Query parameters can be set in three ways:

`setParameter(parameterId, value)`::
Sets a single parameter by its name.

`setParameters(parameters)`::
Sets multiple parameters at once via an object whose keys correspond to
the parameter names.

`setAccessParameter(accessParameter, value)`::
Sets an access parameter (e.g. `'user'`, `'topic'`, `'accessedObject'`) to
a semantic element or a list of semantic elements.

[source,js]
```

query.setParameter('minRating', 4) query.setParameters({ manufacturer: 'ACME', category: 'electronics' })

```
== Executing a Query

=== perform()

The query is executed with `perform()`.
The method optionally accepts a search string and a parameter object that
overrides any previously set parameters.
The two arguments can be passed in any order.

[source,js]
```

const result = query.perform('PowerMole', { manufacturer: 'ACME' })

```
If you only want to pass parameters, omit the search string:

[source,js]
```

const result = query.perform({ manufacturer: 'ACME' })

```
[NOTE]
====
Missing parameters result in a runtime error.
There is no implicit deactivation of parameters that are not passed.
====

=== findElements()

Since `$k.GQLQuery` inherits from `$k.Query`, `findElements()` is also
available.
The method returns an unsorted array of all semantic elements contained in
any of the `RETURN` fields.

[source,js]
```

const elements = query.findElements({ manufacturer: 'ACME' })

```
[NOTE]
====
When using `findElements()`, essential advantages of GQL are lost: the
structured result table, the explicit sort order, and projected expression
values are no longer available.
`findElements()` is therefore only suitable for compatibility with other
query types that use the same interface.
For GQL-specific evaluation, use `perform()`.
====

== Evaluating the Result

The result is returned as a `$k.GQLQueryResult`.

=== Columns

`columns()`:: Returns an array of `$k.GQLQueryResultColumn` objects.
`captions()`:: Returns an array of column names as strings.
```

```
Each column (`$k.GQLQueryResultColumn`) has the following methods:

`index()`:: Returns the zero-based column index.
`caption()`:: Returns the name of the column.
`valueType()`:: Returns the type of the column values.

=== Rows

`rows()`:: Returns an iterator over the result rows.
`size`:: Number of result rows.

Each row (`$k.GQLQueryResultRow`) has the following methods:

`values()`:: Returns the row values as an array.
`bindings()`:: Returns the row values as an object whose keys correspond
to the `RETURN` field names.
`structuredBindings()`:: Like `bindings()`, but field names containing
dots are converted into nested objects (e.g. `p.name` becomes `{p: {name:
"..."}}`).
`size`:: Number of columns in the row.

=== Value Types

Values are returned as the corresponding JavaScript objects:

Nodes::
Nodes are returned as semantic elements.

Edges::
For edges, the type depends on whether a relation index was used.
Either a `$k.Relation` or a `$k.VirtualRelation` is returned.

Attribute values::
Attribute values are returned in the same way as with direct access via
`attributeValue()`.

== Example

Execute the registered query `products` and output the columns and rows as
text.

[source,js]
```

const result = $k.Registry.query('products').perform({ manufacturer: 'Drill Corp' }) const report =

new $k.TextDocument() report.println(`Number of results: ${result.size} rows`) report.println(`Columns:`) for (const column of result.columns()) { report.println(`${column.index()}. ${column.caption()}: ${column.valueType()}`) } report.println(`Rows:`) for (const row of result.rows()) { report.println(row.values().map(v ⇒ JSON.stringify(v)).join(',')) }

```
:leveloffset: 4

<<<
:leveloffset: 2

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//    [{name-of-variable}]
//    |===
//    ....
//    |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#ef8372f0-a073-4442-ba0e-ea451155ecd0]
= Folder and registration

Along with the objects and their properties, we also build a variety of
```

```
other elements in a typical project: we define, for example, queries and
imports/exports, or write scripts for specific functions. Everything that
we build and configure can be organized in folders.

The folders are shared with everyone else working on the project. If we do
not wish to do so, we can file things in the private folder, for example
for test purposes. This is only visible for the respective user.

A special form of the folder is the collection of semantic objects, in
which we can file objects manually, for example for processing at a later
date. To do so, we simply move them to the folders using drag and drop,
and there are also operations to, for example, define result lists in
folders.

The moment we delete one of these objects within the Knowledge Graph, it
is also deleted from the collection. If a semantic element is removed by
clicking on _Remove from folder_, it is only removed from the collection
but still exists within the Knowledge Graph. If the actions _Delete_ or
_Delete selected elements_ or _Delete all elements inside the folder_ is
used, the semantic element actually is deleted and from the Knowledge
Graph and therefore is not accessible anymore within the collection.

[WARNING]
The action _Remove from folder_ has different functionalities, depending
on the context of use: In the case of folders containing import mappings,
the action _Remove from folder_ actually means completely deleting the
respective import mapping!

In the case of collections of semantic objects with more than 100 entries,
for reasons of performance, no determination of the table configuration
that best suits the content occurs. We can, however, request this by means
of the context menu function _Determine configuration of the object list_
when necessary.

== Registration

Queries, scripts, etc. can call each other (a query can be integrated into
another query or into a script, while, in turn, a script can be called
from a search pipeline). There are registration keys for this purpose,
with which we can equip queries, import/export mappings, scripts and even
collections of semantic objects and organizing folders to ensure they
provide other configurations with a functionality. The registration key
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/folder-and-
registration/gfx/tagComponent.png[height=24,width=24] must be distinct.
Everything that has a registration key is automatically added to the
```

```
_Registered objects_ folder, or in the subfolder that corresponds to its
type.

== Move, copy, delete

Let us assume we have a folder called "Playlist functions" in our project.
This might contain an export, some scripts and a structured query "similar
songs", which we would like to use in a REST service. The moment we give
the structured query a registration key, it is added to the folder
_Registered objects_ (_Technical_ section). This means the structured
query "similar songs" appears in the folder _Registered object_ under
_Query_. It also remains there when we remove it from our project
subfolder "Playlist functions". If we remove the registration key, the
query will automatically disappear from the registry.

The basic principle when deleting or removing: Queries, imports, scripts
can be in one or several folders at the same time, and at least one folder
must contain them. Only when we, for example, remove our query from the
last folder will it actually be deleted. Only then does i-views also
request a confirmation of the delete action. The same applies for removal
of the registration key.

If we wish to delete the query in one step, regardless of the number of
folders that contain it, we can only do this from the registry.

== Folder settings

We can define quantitative limits for query results, folders and object
lists (lists of the specific objects in the main window of the Knowledge
Builder when an object type is selected on the left-hand side) in the
folder settings. _Automatic query up to the number of objects_ specifies
up to which number of objects the contents of the folders or the object
lists are shown without any further interaction by the user. If the limit
set there is exceeded, the list initially remains empty, and the message
"Query not executed" appears in the status bar. Executing a search without
an input in the input line shows all objects, unless the second limit has
been exceeded: _Maximum number of query outputs_ or _Maximum number of
outputs in object lists_. In this case the result will also be empty and
the status bar will show a corresponding message. The search then has to
be narrowed down to yield a result.

:leveloffset: 4

<<<
:leveloffset: 2
```

```
// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#71d463ef-5578-4a03-9f98-6fec1ce781c6]
= Import and export

By mapping data sources we can import data to i-views from structured
sources and export objects and their properties in structured form. The
sources can be Excel/CSV tables, databases or XML structures.

The functions for import and export overlap to the most part and are
therefore all available in a single editor. In order to access functions
for import and export, it is first necessary to select a folder (e.g. the
working folder). There the "New mapping of a data source" button can be
used to select a data source for the import or export.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/gfx/141_ImportExport.png[height=148,width=924]
```

```
Alternatively, you can find the button on the "TECHNICAL" tab under
"Registered objects" > "Mappings of data sources".

The following interfaces and file formats are available for import and
export:

* CSV file
* Excel file
* XML file
* JSON file
* LDAP
* Elasticsearch
* IAS Knowledge Model
* MySQL interface
* ODBC interface
* Oracle interface
* PostgreSQL interface

The following section uses a CSV file to describe how to create a table-
oriented import/export.

:leveloffset: 4
:leveloffset: 3

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
```

```
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#7d39a20a-dedc-4225-9927-107fa3f331d0]
= Mapping of data sources

CSV files are the default exchange format for spreadsheet applications
such as Excel. CSV files consist of individual rows of plain text in which
columns are separated by a fixed, predefined character such as a
semicolon.

== Principle of operation

Let's use a table with songs as a first example: When the table is
imported, we would like to create a new, specific object of the type song
for each line. The contents of columns B to G become attributes of the
song, or relations to other objects:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/142_ImportTable.png[height=700,width=910]

Using the song as a basis, we build up the structure of attributes,
relations and target objects that should be created by the import (left-
hand side). An object of type song is created this way for row 18, for
example, with the following attributes and relations:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/143_ImportMapping_3.png[height=529,width=772]

We can, however, also decide to distribute the information from the table
in a different way, for example allocate the year of release and artist to
the album, and in turn the genre to the artist. A row still forms a
context, however this does not mean it must belong to exactly one object:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
```

```
export/mapping-of-data-
sources/gfx/144_ImportMappingDifferent.png[height=539,width=995]
Everywhere that we build up new, specific objects and relation targets in
our example, we must always specify at least one attribute for this
object, in this case the respective name attribute that allows us to
identify the corresponding object.

== Data source -- selection and options

Once we have selected the __New mapping of a data source__ button, a
dialog opens which we must use to specify the type of data source and the
mapping name. If we have already registered the data source in the
Knowledge Graph, then we will now find it in the selection menu at the
bottom.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/145_CreateImportMapping.png[height=630,width=930]

By pressing "OK" as confirmation, the editor for the import and export
opens. We can specify the path of the file we wish to import under "Import
file". Alternatively, we can also select the file using the button to the
right of it. As soon as the file has been selected, the column headings
and their positions in the table are exported and shown in the field at
the bottom right. The __Read from data source__ button can read out the
columns again in the event of any changes to the data source. The column
"Mappings" shows us the respective attribute to which the respective
column of the table is mapped later on.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/146_ImportMappingFile.png[height=678,width=1096]

The structure of our example table corresponds to the full default
settings, so that there is nothing else to factor in under the menu item
__Options__. CSV files can, however, exhibit structures that are very
different, which must be factored in using the following setting options:

__Encoding__: The character encoding of the import file is defined here.
This provides ascii, ISO-8859-1, ISO-8859-15, UCS-2, UTF-16, UTF-8 and
Windows-1252 for selection. If nothing has been selected, the default
setting that corresponds to the operating system in use is applied.

__Line separator__: In most cases, the setting "detect automatically",
```

```
which is also selected by default, is sufficient. However, should the user
establish that line breaks are not being identified correctly, then the
corresponding, correct setting should be selected manually. This provides
__CR__ ("carriage return"), __LF__ ("line feed"), __CR-LF__ and __None__
for selection. The standard used to encode the line break in a text file
is __LF__ for Unix, Linux, Android, Mac OS X, AmigaOS, BSD and others,
__CR-LF__ for Windows, DOS, OS/2, CP/M and TOS (Atari), and __CR__ for Mac
OS up to Version 9, Apple II and C64.

__1st line is heading__: It may the case that the first line does not
include a heading, and the system must be notified of this by removing the
checkmark set by default next to __1st line is heading__.

__Values in cells are surrounded by quotation marks__ is selected so that
the quotation marks are not included in the import when this is not
wanted.

__Identify columns__: Whether the columns are identified using their
heading, the position or the character position must be specified, as
otherwise the table cannot be captured correctly.

__Separator__: If a different separator than the default semicolon is
used, this must also be specified when the column is not identified using
the character position.

Moreover, the following rules apply: If a value in the table contains the
separator or a line break, the value must be placed in double quotation
marks. If the value contains one quotation mark, this must be doubled
(`""`).

== Definition of target structure and mappings
=== The object mapping

We will now start setting up the target structure that should be produced
in the Knowledge Graph. In our example, we are starting with object
mapping of the songs. In order to map a new object, we must press the "New
object mapping" button.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/147_ObjectMapping.png[height=150,width=522]

The next step is to specify the type of object for import.

image:/builds/i-views/documentation/documentation-
```

```
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/148_ObjMapTypeAssignment.png[height=328,width=1095]

There are further specific settings in the options tab of the object
mapping.

**With objects of all subtypes:** If the checkbox is set to "With objects
of all subtypes", the import also includes objects from all subtypes of
"Song". Since this is usually desired, the checkmark is set here by
default.

**Exact type is specified by the following mapping:** If the exact type to
which the object is to be created is identified in the import source, this
can be mapped here via the "New..." button. It must be a subtype of the
type specified in the tab "Mapping".

**Allow multiple objects:** It is possible that the Knowledge Graph
already contains several objects with correspondent identifying properties
(correspondent names). If the import mapping needs to be referred to these
objects, an ambiguity conflict occurs. If you set the checkmark here, the
import for all these objects is going to be performed disregarding the
ambiguity.

If you do not set the checkmark, the import will not be carried out for
the multiple occuring objects and instead the user will be informed that
the importer cannot uniquely identify the object.

=== The attribute mapping / Identifying objects

Now we want to link the information in the table to the object mapping of
the songs. Attributes for individual songs are represented along with
relations. In order to first create the track name for a song in the
mapping, we add an attribute to the object mapping for song. Clicking on
the "__New attribute mapping__" button opens a dialog, which must be used
to select the relevant column from the table to be imported.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/149_AddAttribute.png[height=489,width=804] **
**

As this attribute is the first one we created for the object mapping of
songs, it is then automatically mapped to the name of the object, as the
name is usually the most commonly used attribute.
```

```
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/150_AttributeMapping.png[height=679,width=1096]

The first attribute created for an object is also automatically used for
**identification of the object**. Note that for string attributes like the
primary name, the language can be specified when translation layering is
activated. When nothing else is specified, the current language (display
language of the Knowledge-Builder) is automatically used as reference. The
language can be specified within the language tab:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/151_ImpMapLanguage.png[height=376,width=805]

An object must be identified by at least one attribute -- by its name or
its ID, or by a combination of multiple attributes (as with the first and
last name and date of birth of a person) -- it should already exist so
that it can be unambiguously found in the Knowledge Graph. This prevents
unwanted duplicates from being created during import.

[NOTE]
Meta-Attributes at relations can also be imported. Here it is ensured that
both the relation source and the relation target are specified and
identified, otherwise the relation is ignored by the importer.

In the "__Identify__" tab it is possible to subsequently change the
attribute identifying the object, or to add multiple attributes. In
addition, it is possible to specify whether the values should be matched
in a case-sensitive fashion, and the query should return identical values
(without index filter / wildcards). The latter is relevant if filters or
wildcards are defined in the index that specify, for example, that a
hyphen should be omitted from the index. The term would not be found with
a hyphen if the search took place only via the index; in this case, a
checkmark would be needed here so the search only finds the exactly
identical value.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/152_ImpMapIdentifyObject.png[height=217,width=587]

Now we can add further attributes to object mapping that do not need to
```

```
contribute towards identification, e.g. the length of a song -- and this
is once again done via the "New attribute mapping" button. (Please note:
first the object mapping "objects of song" must be selected again.) Now we
select the "Length" column from the table to be imported. This time we
have to manually select the attribute to be mapped to the "Length" column.
The field on the bottom right contains the list of all possible attributes
defined in the schema that are available to us for objects of the "song"
type, among them also the "length" attribute.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/153_ImpMapAttType.png[height=680,width=1096]

**Mapping of translations**

For string attritbutes with translations, e.g. the primary name of
objects, we can define in which language the value needs to be imported.

If an attribute mapping is created for a translated attribute, the import
language automatically is set to the "Current language". The current
language equals the language in which the Knowledge Builder has been
started (which at the same time is the language of the user interface).

If the import needs to be done in another language than the current
language, this can be specified by selecting the tab "Language" and then
by selecting a language of the list, which then becomes the chosen
language for the attribute mapping.

In case of an import source containing several translations of one and the
same attribute (within the same line), these values can be imported within
one import mapping simultaneously.

The simultaneous import of translations for an attribute is done as
follows:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/469_TranslationImport.png[height=720,width=1280]

 For each language, create a separate attribute mapping for the same
attribute, but specify a different import language

 In the "Language" tab for one of the attribute mappings, add the
relevant attribute mappings of the other languages to the field "Mappings
```

```
of other translations of the same attribute"

This prevents from separate attributes being created for each translation
and ensures that corresponding translations are imported altogether at the
same attribute.

=== The relation mapping

Next, we want to map the album on which the song is located. Since albums
are concreate objects in the Knowledge Graph, we need the relation that
connects the song and the album to do this. To map a relation, we first
select the object for which the relation is defined and then click on the
button __Map new relation__.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/154_AddRelation.png[height=181,width=766]

Following that, just like for attributes, we get a list of all possible
relations; and the required relation __is included in__ is naturally
included.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/155_RelationMapping.png[height=680,width=1094]

In the next step, we now have to define where in this table the target
objects come from. A new object mapping is required for the target; this
is created using the "New" button. If the type of the target object is
uniquely identified in the schema, it is copied automatically. If not, a
list of possible object types appears.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/156_RelationMappingTarget.png[height=679,width=1095]

For new object mappings, we then once again have to select the attribute
that identifies the target object etc. This creates the target structure
of the import.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
```

```
sources/gfx/157_MappingHierarchy.png[height=356,width=384]

=== The type mapping

Types can also be imported and exported. Let's assume we want to import
the genres of songs as types.

To map a new type, we choose the "New type mapping" button.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/158_TypeMappingNew.png[height=145,width=515]

Following this, we have to specify the super-type of the new types to be
created, in our example, the super-type would be "Song":

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/159_TypeMappingSpecify.png[height=263,width=1095]

Following that, we have to specify from which column of the imported table
the name of our new types is to be taken:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/160_TypeMappingName.png[height=536,width=1089]

Following that, we still have to specify on the "Import" tab that our new
types are not supposed to be abstract:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/161_TypeMappingNotAbstract.png[height=282,width=629]

If we now want to assign the corresponding songs to their new types, we
have to use the system relation "has object". In older versions of i-views
this relation is called "has individual". As the target we chose all
objects of song (incl. subtypes), which are defined via the Name
attributes in accordance with the Song title column.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
```

```
export/mapping-of-data-
sources/gfx/162_TypeMappingHasObject.png[height=489,width=1095]

If we now import this mapping, we get the desired result. The songs that
already exist in the Knowledge Graph are taken into account by the import
setting "Update or create if not found" and moved under their respective
type so that no object is created twice (see chapter Import behavior
settings). A quick reminder: A specific object cannot belong to several
types at once.

There is another special case. If we have a table in which different types
occur in one column, we can also map this in our import settings.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/163_ImportTableSpecialCase.png[height=61,width=295]

To do this, we count the mappings of objects to which we want to assign
subtypes (in this case "objects of location") and then select the
corresponding super-type on the "Options" tab.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/164_ObjectTypeMapping.png[height=209,width=874]

It is also important not to forget to specify on the "Import" tab that the
type is not supposed to be abstract so that concrete objects can be
created.

[CAUTION]
Assuming Liverpool already exists in the knoledge graph but is assigned to
the type "Location" because it did not have subtypes such as "City" and
"Country" at that time. In this case, Liverpool is **not** created anew
under the type City. Reason: The objects of the Location type are only
identified via the name attribute and not via the subtype.

=== Mapping of extensions

Extensions can also be imported and exported. Let's assume we have a table
that shows the role of a band member in a band:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
```

```
sources/gfx/165_ImportTableExtensions.png[height=81,width=264]

Ron Wood is a guitarist with the Faces and the Rolling Stones, but a
bassist with the Jeff Beck Group. In order to map this, we must select the
object for which an extension was defined in the schema and then press the
"New extension mapping" button.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/166_ObjectTypeExternsionMapping.png[height=179,width=595]

Like an object mapping, an extension mapping queries the corresponding
type. In the schema of the music graph, the "Role" type is an abstract
type. So it is necessary to define in the mapping that the role is to be
mapped to subtypes of the "Role" type (see Type mapping chapter).

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/167_ImportMappingExtension.png[height=123,width=338]

As with objects and types, the relation can be mapped to the extension (or
to the subtypes of an extension).

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/168_ImportMappingExtCompl.png[height=224,width=376]

== Mapping of several values for an object type at an object

If several values are specified for an object type when there is an object
(in our example, there are several "Moods" for each song), then there are
three possible ways the table will look. For two of the three possible
ways, the import must be modified, which is described in the following.

Option 1 -- Values separated by separators: The individual values are
found in a cell and are separated by a separator (e.g. a comma).

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/173_ImportTableCommaSeparated.png[height=122,width=530]

In this case, we go to the mapping of the data source, where the general
```

```
settings are found, and to the __Options__ tab found there. The setting
used to specify separators within a cell is found here in the lower
section. We now only have to locate the corresponding column of the table
to be imported ("Mood") and enter the separator used (",") in the column
__Separator__.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/174_CommaSeparatedMapping.png[height=505,width=877]

Option 2 -- Several columns: The individual values are located in their
own respective column, whereby not every field must be filled in. As many
columns are required as the maximum number of moods there are per song.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/175_ImportTableColumnSeparated.png[height=121,width=626]

In this case, the corresponding relation must be created the same number
of times as there are columns. In this case, the first relation must,
accordingly, be mapped to "Mood1", the second relation to "Mood2" and the
third relation to "Mood3".

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/176_ColumnSeparatedMapping.png[height=502,width=964]

Option 3 -- Several rows: The individual values are located in their own
respective row. Please note: In this case, it is essential that the
attributes that are required for identification of the object (in this
case the track name) appear in every row, as otherwise the rows would be
interpreted as their own respective object without a name, making a
correct import impossible.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/177_ImportTableRowSeparated.png[height=201,width=466]

In this case, no special import settings are required, as the system
identifies the object using the identifying attribute and creates the
relations correctly.
```

```
== Settings of the import behaviour

During the import process, a check is always performed to determine
whether an attribute already exists. "Identify" infers concrete objects
from attributes. When we refer below to "existing attributes", these are
attributes whose value precisely matches the value in the column to which
they are mapped. When we refer to existing objects, we mean concrete
objects that have been identified through an existing attribute.

**Example:** If our Knowledge Graph already contains a song called
"Eleanor Rigby", the name attribute (mapped to the "track name" column in
our import table) is an existing attribute, so the song is an existing
song as long as the song is identified only via the name attribute.

The settings for import behavior allow us to control how the import should
react to existing and new semantic elements. The following table shows a
brief description of the individual settings, while the sub-chapters of
this chapter contain detailed and descriptive explanations.

[{table-name-text}]
|===
|Setting
|Brief description

|Update
|Existing elements are overwritten (updated), no new elements are created.

|Update or create if not found
|Existing elements are overwritten; if none exist, they are created.

|Delete all with same value (only available for properties)
|All attribute values that match the imported value are deleted for the
respective objects.

|Delete all with same type
|All attribute values of the selected type are deleted for the relevant
objects, regardless of the values match or not.

|Delete
|Is used to delete that exact element.

|Create
|Creates a new property/object irrespective of whether the attribute value
or the object already exists.

|Create if type not found (only available for attributes)
```

```
|An attribute of the required type is only created if none of this type
exists.

|Create if value not found (only available for attributes)
|An attribute with this value is only created, if none with this value
exists.

|Do not import
|No import.

|Synchronize
|In order to synchronize the contents for import with the contents in the
database, this action creates all elements that do not yet exist, updates
all elements that have changed, and delete all elements that no longer
exist.

|===

During an import, we have to decide individually for every mapped object,
every mapped relation and every mapped attribute which import settings we
want to use.

[NOTE]
Unlike in other editors of the Knowledge Builder, a setting is neither
"inherited" by the subordinate mapping elements, nor is the import setting
for an object "inherited" by its attributes.

**The import mapping**

If errors occur due to the data, they will be reported according to their
row numbering when the import transaction is done. Please pay attention
that the row numbering of the error message relates to the table shown in
the import preview when clicking on "Show table".

If empty rows exist in the source table, they are filtered out by the
import mechanism. Therefore pay attention that in case of empty rows, the
row numbering of source table differs from the row numbering of the import
mechanism (including table preview).

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/178_0_ImportBehaviour.png[height=460,width=814]

=== Update
```

```
If this setting is applied to an **attribute**, it ensures that the value
from the table overwrites the attribute value of exactly one existing
attribute. No new attributes are created with this setting. If the object
has more than one attribute value of the selected type, no value is
imported.

If you use the "Update" setting for an identifying attribute while using
the "Update or create if not available" setting for a corresponding
object, the error message "Attribute not found" appears, if the
identifying object is not available in i-views.

If "Update" is applied to an **object**, this setting ensures that all
properties of the object can be added or changed by the import. New
objects are not created.

Example: Let's assume we keep a database of our favorite songs. We have
just received a list with songs that contain new information. We want to
get this information into our database but prevent songs that are not our
favorite songs from being imported. We use the "Update" setting to do
this.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/178_ImpOptUpdateBefore.png[height=335,width=639]

__The song "About A Girl" is already available in the Knowledge Builder.
__

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/179_ImpOptUpdateTable.png[height=41,width=373]

__The import table contains information on the length, rating and creator
of the song.__

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/180_ImpOptUpdateMapping.png[height=315,width=804]

__For Song objects we specify that they are supposed to be updated. All
attributes, relations and relational targets receive the import setting
"Update or create if not available yet".__
```

```
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/181_ImpOptUpdateAfter.png[height=402,width=639]

__The result: The song has been updated and has received new attributes
and relations. Already existing properties have been updated (value).
__

=== Update or create if not found

This import setting is required in most cases and is therefore set as the
default setting. If elements already exist they will be updated. If
elements do not exist yet they are created in the database.

=== Delete all with same value

This import setting is only available for properties (relations and
attributes) and is only used when the import setting "Delete" cannot be
used for deleting. "Delete" does not function for deleting when a relation
or an attribute occurs on an object several times with the same value. If
an attempt is made nonetheless, an error message appears. For example, the
song "About A Girl" may have been linked to the band "Nirvana" using the
relation "has author" by mistake.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/182_ImpOptDelWSameValue.png[height=405,width=639]

In cases like this, the import setting "Delete" does not have an affect,
because due to multiple occurrences, it does not know which relations it
is supposed to delete. In this case, "Delete all with the same value" must
be used.

=== Delete all of same kind

This import setting is used if all attributes, objects or relations of a
type are supposed to be deleted, irrespective of existing values. In
contrast to this, the settings "Delete" and "Delete all with identical
value" take the existing values into account. Only the elements of those
objects that occur in the import table are deleted.

Example: We have an import table with songs and the duration of the songs.
We see that the duration differs in many cases and decide to delete the
duration for these songs to make sure we do not have any incorrect
```

```
information.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/183_ImpOptDelSameKTable.png[height=104,width=353]

__For most songs, the duration in the import table differs...__

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/184_ImpOptDelWSameKBefore.png[height=126,width=556]

__... from the duration of the songs in the database.__

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/185_ImpOptDelWSameKMapping.png[height=187,width=964]

__For the attribute "Duration" we use the import setting "Delete all of
the same type".__

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/186_ImpOptDelWSameResult.png[height=125,width=447]

__After the import, all attribute values of the attribute type duration
have been deleted for these 4 songs.__

=== Delete

The import setting "Delete" is used to delete exactly the one object/
exactly the one relation/exactly the one attribute value. If none or
several objects/relations/attribute values match the elements for import,
an error message about this appears and the elements concerned is not
deleted.

=== Create new

This import setting creates a new property/a new object irrespective of
whether the attribute value or the object already exists. Sole exception:
If a property may only occur once (observe the setting "May have multiple
occurrences" for the attribute definition), then the new attribute is not
```

```
created and an error message appears noting this.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/187_ImpOptCreateNewAttVal.png[height=118,width=1010]

=== Create if type not found

This import setting is only available for attributes. A new attribute
value is only created when the corresponding attribute does not yet have a
value. The values do not have to be the same; what matters is that one
value or another exists, or does not exist, for the corresponding
attribute type. The simultaneous import of several attribute values to one
attribute type is not possible, as in this case it is not possible to
decide which of the attribute values should be used.

Example: Assuming that we have an import table that contains the musicians
with their alias names. A number of musicians also have several alias
names. In this case, we cannot use the setting "Create type if not found",
because then all musicians with several alias names would not be given
one.

=== Create if value not found

This import setting is only available for attributes. A new attribute
value is only created if the object does not yet have this value for the
corresponding attribute.

Example: Let's take again the import table that includes musicians wih
their alias names. Here we can use the setting "Create value if not
found", because then the musicians with several alias names can get all
these alias names.

=== Do not import

The import setting "Do not import" allows us to specify that an object or
a property should not be imported. This is useful when a mapping has
already been defined and we want to use it again, however do not want to
import specific objects and properties again.

=== Synchronize

The import setting "Synchronize" should be used with caution, because it
is the only import setting that not only affects the objects and
properties in i-views that have values that match those in the import
```

```
table, but also extends to all elements of the same type in i-views. When
an import table is synchronized with i-views, in principle this means that
after the import, the result should look exactly the same as it does in
the table.

[CAUTION]
If objects of one type are synchronized, all objects of this type that are
not in the import table are deleted. The objects that exist are updated
and the objects that are not in i-views are created as new objects.

Example: We would like to synchronize the music fairs in i-views (at the
left) with a table with the fairs and their date (at the right):

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/188_ImpOptSyncObjList.png[height=126,width=822]

For objects of the "Fair" type, we select the import setting
"Synchronize;" for the individual attributes __Name__ and __Date of fair__
the import setting "Update or create if not found" is used:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/189_ImpOptSyncMappingOpt.png[height=168,width=966]

The attribute name is the identifiable attribute of fair. There is no name
for the object Music fair 2015 in the import table. If we import the table
this way, an error message is output:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/190_ImpOptSyncErrorMessage.png[height=67,width=766]

After the import, we now see that the import caused two objects to be
omitted that did not have a counterpart in the import table. The date was
updated for Music fair 2016:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-of-data-
sources/gfx/191_ImpOptSyncResult.png[height=134,width=425]

When **attributes** are synchronized, the following applies: When an
```

```
existing attribute is not given a value by an import, it is deleted for
the corresponding object of the import table. If the existing attribute
has a different value to the import table, it is updated, even when it is
allowed to occur several times. If the attribute does not yet exist, a new
one is created.

When **relations** are synchronized, and they are not given a value, they
are deleted for the corresponding object. If the existing relation has a
different value to the import table, it is updated. If the target object
does not yet exist in the database, a new one is created, provided that a
corresponding import setting has been assigned to the target object. If
the target object cannot be created as a new one, because, for example,
the import setting "Update" was assigned, an error message appears
notifying us that the target object was not found and will not be created.

:leveloffset: 4
:leveloffset: 3

[#3349a36f-3f36-4ddf-a484-070a6e001ff2]
= Configuration of data sources

== Table columns

When it comes to mapping database queries, the columns that are available
for import are specified by the database tables and/or the Select
statement. When mapping files, it is possible adopt the columns with the
"Read from data source" button from the file. But you can also specify
them manually. In that case you can choose whether to create a standard
column or a virtual property.

If you want to export from the Knowledge Graph you have to enter the
columns manually. You can export only standard columns, not virtual
columns.

**Virtual table column / virtual property**
Virtual columns are additional columns that allow you to use regular
expressions to transform the contents we find in a column of the table to
be imported. Example: Let's assume that "a.d." is appended to all the
years in our import table. We can correct this by creating a virtual
column that adopts only the first 4 characters from the year column.

We can also define virtual properties during export.

We simply write the **expressions** into the column header (into the name
of the column). During the process, partial strings enclosed in pointy
brackets <...> are replaced according to the following rules, with __n,
```

```
n1, n2, ...__ representing the contents of other table columns with the
column number n.
[%header, cols="1,2a,1a,1a,1a"]
|===
|Expression
|Description
|Example
|Input
|Output

.2+|<n **p** >
.2+|Print output of content of column n
.2+|Hits: <1p>
|1 (integer)
(string)
|Hits: 1
|"none"
|Hits: "none"

|<n **s** >
|Output of string in column n
|Hello <1s>!
|"Peter"
|Hello Peter!

|<n **u** >
|Output of string in column n in upper case
|Hello <1u>!
|"Peter"
|Hello PETER!

|<n **l** >
|Output of string in column n in lower case
|Hello <1l>!
|"Peter"
|Hello peter!

.3+|<n **c** start-stop>
.3+|Partial string from position start to stop from column n
|<1c3-6>
|"Columns"
|lumn
|<1c3>
|"Columns"
|mns
|<1c3\->
```

```
|"Columns"
|lumns

.4+|<n **m** regex>
.4+|Test whether the content of column n matches the regex regular
expression. The following expressions are only evaluated if the regular
expression applies.
.2+|<1m0[0-9]>hi
|01
|hi
|123
|(blank)
.2+|<1m$>test
|(blank)
|test
|123
|(blank)

.2+|<n **x** regex>
.2+|Test whether the content of column n matches the regex regular
expression. The following expressions are only evaluated if the regular
expression does **not** apply.
.2+|<1x0[0-9]>hello
|01
|(blank)
|123
|hello

.2+|<n **e** regex>
.2+|Selects all hits for regex from the contents of column n. Individual
hits are separated by commas in the result.
|<1eL+>
|HELLO WORLD
|LL,L
|<1e\d\d\d\d>
|02.10.2001
|2001

|<n **r** regex>
|Removes all hits for regex from the contents of column n
|<1rL>
|HELLO WORLD
|HEO WORD

|<n **g** regex>
|Transmits the contents of all groups of the regular expression
```

```
|<1g\+(\d+)\->
|+42-13
|42

.4+|<n **f** format>
.4+|Formats numbers, date and time specifications from column n according
to the "format" format specification
.2+|<1f#,0.00>
|3.1412
|3.14
|1234.5
|1234.50
|<1fd/m/y>
|1 May 1935
|1/5/1935
|<1fdd/mmm>
|1/5/1935
|01/May

// Workaround for layout bug (incomplete line below previous row)
|
|
|
|
|

|===

Table columns can also be referenced independently from their column
number by using specially defined identifiers. The advantage in this case
is that the allocation is not lost if the column order is changed in the
import table.

The identifier for the relevant column of the import table is entered in
the column with the heading __Identifier__ in the column definition table.
These columns are referenced by creating a virtual table column that is
given the identifier as its table column heading (see example 2).
[%header, cols="1a,2a,1a,1a,1a"]
|===
|Expression
|Description
|Example
|Input
|Output

|<**$** name **$expr**>
```

```
|Reference to a column by means of a unique column identifier __name__ and
subsequent transformation by means of the expression.The **$** characters
are a functional component of the identifier syntax.
|<$Name$u>
|"mp3"
|MP3

|===

For more information on how to use regular expressions (regEx), see
link:https://regex101.com/[https://regex101.com/].

**Example 1: Use of expressions (reference via column number)**

Let's assume we have an import table containing concrete objects without a
name. However, we want these objects to be modeled as separate objects in
our data model. Example: for a load point, column 88 contains its main
value, which is torque. So we enter the expression __load point <88s>__ as
the definition of our virtual column that will represent the name of this
load point. The resulting name for a load point with a torque of 850 would
therefore be "load point 850".

We can also use the virtual property to create a username consisting of
the first 4 letters of the first name and the last name. If the person is
named Maximilian Mustermann and we define the virtual column with the
relevant expression <1c1-4><2c1-4>, the result is "MaxiMust".

The virtual property can also be used to create an initial password for a
user during import. The expression could be __Pass4<2s>__. The resulting
password for Maximilian Mustermann would be "Pass4Mustermann".

A rather extensive example shows how the virtual property can be used to
assign objects to the correct direct top-level group:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-data-
sources/gfx/192_TableColumnPreview.png[height=382,width=1075]

The three right columns are virtual columns.

__<1mCD>:__ The number of the top-level group of the object is only
written to the first of the virtual columns if the term "CD" (for compact
disc) occurs in the first column for the object.

__<2c1-3>000:__ The number to be written to the column consists of the
```

```
first three characters of the second column and three zeros.

__<1xCD><1xMD>:__ Only if the first column for the object does not contain
"CD" or "MD", the content is written to the column.

__<2c1-4>00:__ The number to be written to the column consists of the
first four characters of the second column.

__Playlist Summer 2019:__ This expression is written to the column for all
objects.

**Example 2: Use of individual identifiers (in combination with regular
expressions)**

In the following example, the contents for the **Media** column are
transformed into upper-case letters and into filename extensions by means
of virtual columns: Column 6 uses a reference per column number, column 7
uses a reference per column identifier.

To set up columns with virtual values, do as follows:

1. Enable the editing of columns first
2. Add the identifier name for the column (the identifier "media" always
will stick to the column with the title "Media")
3. Click on the "Add column" button
4. Choose the virtual property
5. For the heading, enter the column identifier in combination with the
regular expression
6. To ensure that the current data is loaded, click on "Read from data
source"
7. Click on "Show table" to see the result

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-data-
sources/gfx/193_VirtualValueIdentifierWithRegex.png[height=618,width=948]

A click on the "Show table" button shows the preview with the transformed
column entries:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-data-
sources/gfx/194_VirtualValueIdentifierBefore2.png[height=440,width=1110]

The following figure shows the effect of swapped columns in an import
```

```
table: If only column numbers are used like in < **1** u>, the wrong
column is accidently transformed; if an **identifier** is used with a
downstream regular expression like in <$ **media** $m(mp3|ogg)>, the
content is still referenced correctly and therefore transformed into the
correct virtual value:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-data-
sources/gfx/195_VirtualValueIdentifierAfter2.png[height=446,width=1105]

**Functioning and sequence of regular expressions**

The previously shown regular expression work as follows:

* The regular expression "m(mp3|ogg)" matches all entries either
containing "mp3" or "ogg".
* The letters "*." outside the parentheses simply will be added to the
result in order of their appearance.
* The regular expression <$media$l> transforms all letters into lower case
letters.

For the sequence of the regular expressions, it is important to set the
__filtering__ regular expression before the __transforming__ regular
expression:

<$media$m(mp3|ogg)> filters the entries which will be transformed by
<$media$l> afterwards.

The complete regular expression <$media$m(mp3|ogg)>*.<$media$l> returns
the intentded result, whereas another sequence of the expressions
*.<$media$l><$media$m(mp3|ogg)> result into all entries being transformed.
Because the transforming expression works like an immediate output, the
filtering expression it is not obeyed anymore, leading to the rather
unusual music filename extensions *.lp, *.cd or *.md:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-data-
sources/gfx/196_VirtualValueRegexOrderMixed.png[height=382,width=1075]

== Configuration of further table oriented data sources

**Databases**

The database, user and password must be specified in the mapping for a
```

```
PostgreSQL, Oracle or ODBC interface.

**Database specification**

The database specification consists of the name of the host, the port, and
the name of the database. The syntax is:
[%header, cols="1a,2a"]
|===
|Database system
|Database specification

|PostgreSQL
|
+++hostname:port_database+++

|Oracle
|
+++//hostname:[port][/databaseService]+++

|ODBC
|Name of the configured data source

|MySQL
|Separate configuration of database and host name

|===

**Configure user name and password**

The user name and password are specified as stored in the database. Under
the Table option it is possible to specify the table to be imported.
However, for import there is also the option of going to the "Query"
option and formulating a query that specifies which data are to be
imported.

**Encoding**

In case of PostgreSQL mapping, it is possible to specify the encoding on
the "Encoding" tab.

**Special requirements of the Oracle interface**

The function for direct import from an Oracle database requires that
certain runtime libraries are installed on the computer performing the
import.
```

```
What is required directly is the "Oracle Call Interface" (OCI), and it is
required in a version that, according to Oracle, matches the database
server to be addressed. That means that the OCI in version 11 must be
installed on the importing computer in order to address an Oracle 11i
database. The easiest way to install the OCI is to install the "Oracle
Database Instant Client". The "Basic" package version is sufficient. The
client can be obtained from the company operating the server, or from
Oracle after registering at
http://www.oracle.com/technology/tech/oci/index.html.

After the installation, it must be ensured that the library can be found
by the importing client, either by placing it in the same directory or by
defining environment variables that match the relevant operating system
(documented for the OCI).

Depending on the operating system on which the import will be executed,
further libraries are necessary, and these are not always installed.

* MS Windows: next to the required "oci.dll", two further libraries are
required: advapi32.dll (extended Windows 32 Base-API) and mscvr71.dll
(Microsoft C Runtime Library)

Apart from the XML import/export, all imports/exports are table-based and
differ only in terms of the configuration of the source. For a description
of a table-oriented display, you can consult the link:http://www.k-
infinity.de/doku/4.0/dispatch_getFullDocument.do?dmid=ID564579_534626329#d
ocPartID568281_31039691-chapter[Example of the CSV file].

== Mapping of an XML file

The principle of XML files is to make the different details for a record
explicit by means of tags (<>) (and not by means of table columns).
Accordingly, tags are also the basis for display when XML structures are
imported to i-views.

Example: Let's assume that our list of songs is available as an XML file:

[source,xml]
```

<?xml version="1.0" encoding="ISO-8859-1"?> <Contents> <Album type="Oldie"> <Title>Revolver</Title> <Song nr="1"> <Title>Eleanor Rigby</Title> <lengthSec>127</lengthSec> <Artist>The Beatles</Artist> <Topic>Mental illness</Topic> <Mood>Dreamy</Mood> <Mood>Reflective</Mood> </Song> [...] </Album> [...] </Contents>

```
If we want to import this XML file, we choose the "XML file" data source
```

```
when selecting the type, which causes the editor for the import and export
of XML files to open. Even the specification of the file location is
different than in the editor for CSV files. We can now choose between a
local file path and specification of a URI.

**JSON preprocessing** makes it possible to convert a JSON file to XML
before the actual import.

You can choose **Transform with XSTL** if you want to convert the XML data
from the selected XML file to different XML data before the import, for
example in order to change the structure or further separate individual
values. Use the "Edit" button to open the XML file, where you can then
define the changes by means of XSLT.

Once the file has been selected, use the "Read from data source" button to
read out the XML structure, which is then displayed in the right-hand
window.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-data-
sources/gfx/197_XMLImportMapping.png[height=927,width=1160]

We want to import the individual songs on our list. So we create a new
object mapping and use the "Map to" button to select the __<Song>__ tag.
In contrast to a CSV import, where only the attribute values have an
equivalent in the CSV table and where an individual row represents an
object, which means that only the attribute values need to be mapped,
semantic objects are here mapped by the XML structure. Therefore a
corresponding tag of the XML file must be specified for each of the
objects to be mapped.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-data-
sources/gfx/198_XMLImpMapTypeSel.png[height=590,width=865]

As our example shows, the tags are not always unambiguous without context:
<Title> is used for both album titles and song titles. The object type
only becomes clear in combination with the surrounding tag. Often the
context of the XML structure and the context of the mapping hierarchy are
synchronous: As we have already specified that the objects should be
mapped to the <Song> tag, the XML structure makes clear which <Title> tag
we actually mean when we map <Title> with the name attribute of songs.
Where the mapping hierarchy and the tag structure are not parallel, we can
use XPath to form strings in the XML import in addition to the tags
```

```
occurring in the XML file.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-data-
sources/gfx/199_XMLImpMapAttSel.png[height=573,width=865]

As with the CSV import, it is necessary to use the "Identify" tab to
specify for object mapping which attribute values should be used to
identify the object in the Knowledge Graph. The first created attribute
for an object is once again used automatically as the identifying
attribute.

**Options with XPath expressions**

Let's assume we only want to import songs from albums with the "Oldie"
music style. In our XML document, the information for the music style is
specified directly in the album tag under __type="..."__. That means we
have to use the editor to define an XPath expression describing the path
in the XML document that contains only songs from oldie albums. The right-
hand lower section of the editor contains a field for adding XPath
expressions.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-data-
sources/gfx/200_XMLImpMapAddXPath.png[height=155,width=575]

The correct XPath expression is: `//Album[@type="Oldie"]/Song`

Explanation in detail:

* `//Album`: Selects all albums; their position in the document is
irrelevant.
* `Album[@type="Oldie"]`: Selects all albums of the "Oldie" type
* `Album/Song`: Selects all songs that are sub-elements of albums.


We can now use this expression to define an equivalent for the object
mapping of songs.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-data-
sources/gfx/201_XMLImpMapAssignXPath.png[height=283,width=865]
```

```
XPath also offers many other useful selection functions. We can, for
example, select elements by their position in the document, use
comparative operators, and specify alternative paths.

**Basic tips for the XML import**

* Use one absolute path.
* Express all other paths relatively to the absolute path.
* An incremental import only is possible if no cross-references are going
to be imported.
If so, define the node with the absolute path as a partitioning element
(see option on the second tab of the import mapping).
* If the structure branches out into the depth, an import mapping going
from deeper level towards upper level is recommended, since there is only
one parent element instead of several child elements.
* In case of more complex XML documents, it can be beneficial to import
all objects including their identifying attributes first and the
relationships in a second step. This ensures that all objects can be found
for building relationships.

**Alternative: XML import mapping for RDF files**

If the schema in the semantic network is too specific for the existing RDF
file __or__ if the RDF file is too specific __or__ the rdf schema is
missing so that it cannot be imported by the import mechanism correctly,
we can use the XML import mapping for specified import.

In most cases, we will need to use XPath expressions for dedicated value
assignment. Pay attention that for the XML import mapping, an interactive
step-by-step import is not available.

[NOTE]
For Xpath expressions, the namespace (built up on to the qualifier) is not
considered by the system for import mapping.
[cols="1a,1a,1a"]
|===
|Input RDF-XML
|XPath
|Meaning

|
|+++//+++
|Top-level of the RDF

|
|+++../+++
```

```
|One level above

|
|+++../../xyz+++
|Two levels above, from there the node below called "xyz"

|----
<rdf:label>
```

|/label/ |Tag "label"

|---- <rdf:prefLabel xml:lang="en"> Example </rdf:prefLabel>

```
|prefLabel[@lang="en"]
|Node with attribute and certain attribute value. Output = "Example".

|
|ancestor::termEntry/attribute::id
|Superordinate node on any level with name ("termEntry") and attribute
("id")

|
|/myparent/mychild[text()]
|Text between certain tags

|===

== Mapping of JSON files

// TODO: translate

:leveloffset: 4
:leveloffset: 3

[#0d382faa-b9b3-41d2-b6f2-86071968ce0c]
= Further options, log and registry

== Further options at the import

In the "Options" tab, the following functions are available for selection
irrespective of the data source:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-options/gfx/202_ImportOptionsTab.png[height=130,width=558]
```

```
**Import in one transaction** : This is slower than an import with several
transactions and should only be used if a conflict would occur during an
import with several transactions because many people are working in
Knowledge Builder at the same time or because you want to import data
where it matters that individual pieces of data are not viewed separately
from each other.

**Example 1:** Every hour, an import is executed with the machine load
status. The combined load values must not exceed a certain value as that
could result in a power failure. To ensure this rule can be taken into
account (e.g. by means of a script), all values must be viewed jointly and
then imported.

**Example 2:** An import is executed with persons of which no more than
one person may have a master key because only one master key exists. The
import must also be performed in one transaction here because several
transactions could result in missing the error that the attribute for the
master key has been set for two persons.

**Use several transactions** : Default setting for fast import.

**Journaling** : Journaling should be used if very large amounts of data
are deleted or modified in one import. The changes or deletions for these
entries are only to be made to the index after 4,096 entries (the figure
is variable). This speeds up the import because the index does not have to
be used for every single change/deletion. Instead, these changes are
copied to the index after a maximum of 4,096 changes.

**Update metrics** : Metrics are supposed to be updated if the import
significantly affects the number of object types or property types, that
is, if a large number of objects or properties of a type are added to the
Knowledge Graph. If the metrics were not updated, this could negatively
affect the performance of searches in which the corresponding types play a
role.

**Trigger activated** : You can use this checkmark to determine if the
trigger is supposed to be activated or not during import. If you wish to
apply one trigger but not another one, you have to define two different
mappings with the corresponding semantic elements. For information on
triggers, refer to the Trigger chapter.

**Automatic name generation for nameless objects** : Enables the automatic
name generation for nameless objects.

If there is a table-oriented source, we can make the following settings:
```

```
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-
options/gfx/203_ImportOptionsDataSource.png[height=238,width=579]

**Import entire table** : Even though it can take longer to import the
entire table at once, it makes sense to select this option if there are
forward references, i.e. if relations are to be drawn between the objects
to be imported. In this case, both objects must already be available,
which is not the case if the table is imported one row at a time.
Furthermore, the progress display is more precise than for importing one
row at a time.

**Import table row by row** : A table should always be imported one row at
time when the table contains no source reference since this procedure
speeds up the import.

**Separators within a cell:** Refer to the chapter Mapping several values
for an object type for an object.

If we have an XML-based data source, the following functions are
available:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-
options/gfx/204_ImportOptionsXMLDataSource.png[height=174,width=580]

**Incremental XML import:** The XML import is performed step-by-step.
These steps are specified by the partitioning element.

**Import DTD:** Imports the document type definition (DTD) **.

== Log

The functions in the "Log" tab allow changes that are made upon import to
be tracked.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-options/gfx/205_ImportMappingLog.png[height=319,width=593]

**Place generated semantic elements in a folder:** If new objects, types
or properties are generated by the import, they can be placed in a folder
in the Knowledge Graph.
```

```
**Place changed semantic elements in a folder:** All properties or objects
with properties that were changed by the import can be placed in a folder.

**Write error messages to a file:** Errors can occur during import (for
example, there may have been an identifying attribute for several objects,
which is why the object could not be identified uniquely). These errors
are displayed in a window following import by default, and the option of
saving the error log is provided. If this is to occur automatically, then
a checkmark can be placed in the box and a file can be specified here.

**Last import / Last export** : The date and time of the last import
performed and the last export performed are displayed here.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-
options/gfx/206_ImportMappingObjectLog.png[height=192,width=577]

The "Log" tab is also available in the case of the individual mapping
objects. When necessary, a category can be entered for log entries here.
Moreover, it is possible to define that the value of the corresponding
object/corresponding property should be written into the error log. This
is not activated by default, in order to avoid revealing sensitive data
(e.g. passwords).

== Registry

The function "Set registry key" can be found under the "Registry" tab, and
can be used to register the data source for other imports and exports.

The function "Link existing source" allows a registered source to be used
again.

"References" shows other places where a data source is being used:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/mapping-
options/gfx/207_ImportMappingRegistry.png[height=588,width=790]

:leveloffset: 4
:leveloffset: 3

// Include language-specific variables
// language specific definitions (EN)
```

```
:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#9abe4234-49a2-44f8-9dc1-91e136e1e978]
= Attribute types and formats

One frequent job of attribute mapping is to import specific data from
concrete objects, for example from persons: Telephone number, date of
birth etc.

For the import of attributes for which i-views uses a specific format
(e.g. date), the entries of the column to be imported must be provided in
a form that is supported by i-views. For example, a string in the form
`abcde...` cannot be imported to an attribute field of the date type; in
this case, no value is imported for the corresponding object.

The following table lists the formats that i-views supports during the
import of attributes. A table value `yes` or `1` is, for example, imported
correctly as a Boolean attribute value (for a correspondingly defined
attribute), while a value such as `on` or similar is not.
[%header, cols="1a,2a"]
```

```
|===
|Attribute
|Supported value formats

|Selection
|The mapping of import to attribute values can be configured with the
"Value allocation" tab.

|Boolean
|The mapping of import to attribute values can be configured with the
"Value allocation" tab.

|File
|It is possible to import files (e.g. images). For this to happen, either
the absolute path to the file must be specified, or the files to be
imported must be in the same directory (or a subdirectory that needs to be
specified) as the import file.

|Date
|
* <day> <monthName> <year>, e. g. 5 April 1982, 5-APR-1982
* <monthName> <day> <year>, e. g. April 5, 1982
* <monthNumber> <day> <year>, e. g. 4/5/1982

The separator between <day>, <monthName> and <year> can be a space, a
comma or a hyphen, for example (but other characters are also possible).
Valid month names are:
* "January", "February", "March", "April", "May", "June", "July",
"August", "September", "October", "November", "December"
* "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct",
"Nov", "Dec".

[NOTE]
Two-digit years are expanded to 20xy (so 4/5/82 becomes 4/5/2082). If
mapping is set to "Freely definable format", the following tokens can be
used: YYYY and YY (year), MM and M (month number), MMMM (name of month),
MMM (abbreviated name of month), DD and D (day)

|Date and time
|For date and time see the corresponding attributes. The date must come
before the time. If the time is omitted, 0:00 is used.

|Color
|Import not possible.

|Fixed point figure
```

```
|Import possible.

|Integer
|
* Integers of any size
* Floats (separated by a point), e.g. 1.82. The figures are rounded during
import.

|Internet link
|Any URL possible.

|Time
|<hour>: <minute>: <second> <am/pm>, e.g. 8:23 pm (becomes 20:23:00)
<minute>, <second> and <am/pm> can be omitted. If mapping is set to
"Freely defined format", the following tokens can be used: hh and h
(hour), mm and m (minute), ss and s (second), mmm (millisecond)

|String
|Any string. No decoding is performed.

|===

**Boolean attributes and selection attributes**

Selection or Boolean attributes can only assume values from a specified
set; for selection attributes this is a specified list, and for Boolean
attributes this is the value pair `yes`/`no` in the form of a clickable
field. When importing these attributes, you can specify how the values
from the import table are translated to attribute values of the Knowledge
Graph. One option is to adopt the values as they are listed in the table;
if they do not correspond to any possible attribute values defined in the
Knowledge Graph, they are not imported. The other option is to specify
value allocations between table values and attribute values, which are
then imported.

:leveloffset: 4
:leveloffset: 3

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
```

```
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#3a7c9a98-1c2b-430e-a2ff-a60d8ad8a4f5]
= Configuration of the export

The export of data from a Knowledge Graph into a table is prepared in the
same editor and in the same way as the import.

. A new mapping is created in a table mapping folder in the main window.
. In the table mapping editor, the file to be generated is specified.

The difference to the import is that the columns are not imported from the
table now but have to be created in the table mapping editor. Since the
import and export editor are one and the same, you first have to select
whether a new column to be created is a __standard__ column or a __virtual
property__. However, virtual properties cannot be used for export.

== Exporting structured queries

It is possible to export the result of a structured query. This procedure
makes sense if only certain objects that have been restricted by a search
are supposed to be exported. Let's assume, for example, we want to export
all bands that have written songs that are more the 10 min long. To do
this, we first have to define a structured query that collects the desired
objects.
```

```
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-the-
export/gfx/208_ExportMappingStructuredQuery.png[height=281,width=1082]

We then access this structured query from the configuration of the export.
To do this, we select the mapping of a query rather than an object mapping
in the mapping configuration header. The structured query can only be
accessed with a registration key.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-the-
export/gfx/209_ExportMappingAddStrucQuery.png[height=149,width=687]

This has the effect that only the results of the structured query are
exported. For these objects, we can now create properties that are to be
included in the export: e.g. the year the band was founded, members and
songs. However, we might not want to export all of the songs of the bands
we have thus compiled but only those songs that also match the search
criterion, which is songs longer than 10 min in our example. To do this,
we can assign identifiers to the individual search conditions in the
structured query. These identifiers in turn can be addressed in the export
definition.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-the-
export/gfx/210_ExportMappingAddStrucQueryIdentifier.png[height=258,width=9
37]

== Exporting collections of semantic objects

Collections of semantic objects can also be exported. These also need a
registration key, which you can set under TECHNICAL > Organizing folder.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-the-
export/gfx/211_ExportMappingAddFolder.png[height=149,width=764]

== Exporting the frame ID

The mapping of the frame ID enables us to export the ID of a semantic
element assigned in the Knowledge Graph. To do this, we simply select the
object, type or property for which we need the ID and then choose the "New
```

```
mapping of Frame ID" button:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-the-
export/gfx/212_ExportMappingAddFrameID.png[height=280,width=783]

We can also decide if we want to output the ID in string format
(ID123_456) or as a 64-bit integer.

== Export via script

Finally, we have one additional powerful tool for the export: script
mapping.

For the export, we have to specify the columns for the properties to be
exported. For the mapping of the individual property, we then can asssign
the output column ("Map to"):

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-the-
export/gfx/169_ExportMappingColumns.png[height=720,width=1280]

The script mapping is, for example, used when we wish to combine three
attributes from the Knowledge Graph to form an ID. However, this may slow
down the export. (In the case of an import, this could be mapped using a
virtual property more easily. The use of virtual properties is explained
in the chapter "Table Columns".)

The following case is another example of the use of a script in the case
of an export. It shows how several properties can be written into a cell
with a separator. In this case, we wish to generate a table which lists
the song names in the first column and all moods for the songs separated
by commas:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-the-
export/gfx/170_ScriptMapping.png[height=222,width=760]

To generate the second column, we require the following script:

[source, js]
```

function exportValueOf(element) { var mood = ""; var relTargets = $k.Registry.query("moodsforSongs").findElements({songName: element.attributeValue("objectName")}); if(relTargets && relTargets.length > 0) { for(var i=0; i < (relTargets.length-1); i++) { mood += relTargets[i].attributeValue("objectName") + ", "; } mood += relTargets[relTargets.length-1].attributeValue("objectName"); } return mood; }

```
The script contains the following structured query (registration key:
"mood ForSongs"):

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-the-
export/gfx/171_ScriptMappingStructuredQuery.png[height=186,width=833]

The expression "findElements" allows us to access a parameter (in this
case "songName") within the query. The "objectName" is the internal name
of the name attribute in this Knowledge Graph.

Within the if-instruction we state that when an element has several
relation targets, these should be shown separated by a comma. After the
last relation target that runs through the loop, there should no longer be
a comma. Even when an element only has one relation target, this is shown
without a comma accordingly.

The result is a list of songs with all their moods, which appear separated
by a comma in the second column in the table:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/configuration-of-the-
export/gfx/172_ScriptExportTable.png[height=344,width=630]


== Export actions for database exports

Mapping the properties of an object for an export into a database takes
place exactly like mapping for an import and all other types of mapping.
The only difference is that the export action has to be specified for the
export. This specifies which type of query is to be executed in the
database. Three export actions are available:

The following actions are available in the selection dialog that opens:

* **Create new data records in table** : New data records are added to the
database table. This action corresponds to an INSERT.
```

```
* **Update existing data records** : The data records are identified via
an ID in the table. They are only overwritten if the value has changed. If
there is no suitable data record, a new one is added. This action
corresponds to an UPDATE.
* **Overwrite table content during export** : All data records are first
deleted and then written again. This action corresponds to an DELETE on
the entire table followed by an INSERT.

:leveloffset: 4
:leveloffset: 3

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#56f48038-5f93-46cd-b5db-72b52d8dbef9]
= RDF Import and Export

RDF is a standard format for semantic data models. With RDF import and
export, you can exchange knowledge graph data with other applications or
```

```
transfer data from one knowledge graph to another.

The import supports the RDF/XML and Turtle formats, while the export
supports RDF/XML.

For further information about the RDF standard, see
link:http://www.w3c.org/rdf[W3C].

== Basics

To reference the contents of an RDF file, a URI is used, which is referred
to as _RDF URI_ in the following. This RDF URI is often composed of a base
URI and an ID, which is referred to as _RDF ID_ in the following:

Example:
[cols="1a,2a"]
|===
|Base URI
|`\http://www.example.org/heavytools#`
|RDF ID
|`mini2000`
|Resulting URI
|`\http://www.example.org/heavytools#mini2000`
|===

URI namespaces can also be named.
Complete URIs can then be specified as a combination of the name and the
RDF ID.

[source,xml]
.Namespaces in RDF-XML
```

<rdf:RDF xml:base="http://www.example.org/heavytools#" xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" xmlns:ht="http://www.example.org/heavytools#">

```
  <ht:excavator rdf:ID="mini2000">
    <rdfs:label>Mini 2000</rdfs:label>
  </ht:excavator>
</rdf:RDF>
```

```
Attribute values are represented as strings.
```

```
When importing into and exporting from a knowledge graph, the strings are
encoded according to XSD data types. For example, a date is represented
according to `xs:date`.

[source,xml]
.Date in RDF-XML
```

<ht:dateOfManufacture>2003-04-05</<ht:dateOfManufacture>

```
The optional language is specified according to RFC 3066.
For example, _German_ is encoded as `de`:
[source,xml]
.Attribute with language specification in RDF-XML
```

<rdfs:label xml:lang="de">Bagger</rdf:prefLabel>

```
For relations, the URI of the relation target is specified. This can be
shortened using the base URI:
[source,xml]
.Relation with shortened URI in RDF-XML
```

<ht:uses rdf:resource="#excavator"/

```
[NOTE]
====
RDF/XML does not allow the use of namespaces with `rdf:resource`. The
following is not a valid URI:
[source,xml]
```

<ht:uses rdf:resource="ht:excavator"/

```
====


[#56f48038-5f93-46cd-b5db-72b52d8dbef9-identification]
== Identifying Objects in the Knowledge Graph

You can identify objects by the following properties:

Attribute _rdf:about_:: Complete URI of the object.
Attribute _rdf:ID_:: Shortened URI. The ID forms a complete URI together
with a base URI.
```

```
Attribute _RDF URI alias_:: Additional URI used for identification during
import.
Frame ID:: Internal, immutable ID of an object.
This can only be used when transferring data from one knowledge graph to a
copy, for example from a development to a production environment.

== Global Settings

You can configure the base URL of the knowledge graph in the global
settings of the Knowledge Builder. It applies to both import and export:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/rdf-import-and-
export/gfx/214_RDFGlobalSettings.png[height=515,width=760]

The entries under "Additional namespaces" apply only to the export and are
used there to shorten URLs.

== RDF Import

You can start the RDF import from the main menu via __Tools > RDF > RDF
Import__.
In the subsequent dialog window, you can select the file to import:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/rdf-import-and-
export/gfx/217_RDFImportFileSelection.png[height=222,width=502]

.Options

Import referenced resources::
If you select this option, all resources referenced in the RDF file are
imported as well.
[NOTE]
Note that the referenced resources may in turn contain further referenced
resources, which could lead to uncontrolled data import.

Ignore HTTP errors::
The Knowledge Builder outputs error messages if the RDF label "namespace"
is missing in the URL of the RDF file. In that case, only the namespace
HTTP URL is considered. Enabling this option suppresses the error
messages.

Identify objects with global URI also by local ID::
```

```
If you enable this option, objects are identified by the RDF ID even if
the base URI does not match. You should only enable this option in
exceptional cases.

=== Configuring the Import

After reading the file to be imported, an overview of the read data is
displayed.
At this point, the data has not yet been imported.

The _Schema changes_ tab lists all planned changes to the schema.
The _Legend_ tab contains an explanation of the symbols used.

==== Manual Mapping
Objects are identified by the properties described in the section
<<56f48038-5f93-46cd-b5db-72b52d8dbef9-identification>>.
You can manually adjust the mapping by selecting an object in the
hierarchy and choosing an object under _Map to_ on the right side.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/rdf-import-and-
export/gfx/218_RDFImportStrategy.png[height=295,width=768]

==== Options

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/rdf-import-and-
export/gfx/219_RDFImportOptions.png[height=164,width=541]

Allow schema changes::
If schema changes are planned, you should enable this option.

Avoid property duplicates::
In RDF files, identification of properties is not common.
To prevent duplicates of properties from accumulating when importing
already existing objects, you can enable this option.
Existing properties of a type are then identified by value (for
attributes) or by relation target (for relations).

References to absent resources::
When importing data from public sources, you can use the `RDF-refersTo`
attribute as a substitute reference for (temporarily unavailable)
dependencies.
This attribute serves for identification of the sources in case of a
```

```
subsequent import.
This is particularly useful when the RDF file contains empty parts with
URLs without type definitions.

Triggers enabled::
By default, trigger functions are not active during the import.
If you still need triggers, you can enable them with this option.

Import qualifiers/namespaces::
This option is useful when reimporting an RDF file that originates from
the same knowledge graph.
If the RDF file has a foreign namespace, you should not use this option.

.Log Options

Create folder with imported objects::
This option allows you to inspect the imported elements after import in a
knowledge graph folder located in the working folder.

With relation targets::
If the RDF file contains new objects with relations whose relation targets
already exist in the knowledge graph, the relation targets will be
included in the folder of imported objects.

.Transaction

Import in a single transaction::
All data is imported in a single transaction.
If transaction conflicts occur, the knowledge graph is not changed.

Use multiple transactions::
This option is recommended if the RDF file contains a large amount of
data.
If transaction conflicts occur, only parts of the file are imported.

==== Additional RDF Import/Export Options

You can also import RDF files via the REST interface using a JavaScript
function.
For further information, see the JavaScript API documentation:
link:https://documentation.i-views.com/{iviews-version}/javascript-
api/$k.RDFImporter.html[https://documentation.i-views.com/{iviews-
version}/javascript-api/$k.RDFImporter.html]

== RDF Export
```

```
You can export the entire knowledge graph or individual elements as an RDF
file.

Entire knowledge graph::
You can start the RDF export of the entire knowledge graph from the main
menu via __Tools > RDF > RDF Export__.

Individual objects::
You can export individual objects by selecting _RDF Export_ in the context
menu of the object.

Elements from a collection of semantic elements::

You can export multiple objects by creating a collection of semantic
elements and selecting _RDF Export_ in the context menu of the collection.

=== RDF Export Settings

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/rdf-import-and-
export/gfx/221_RDFExportOptions.png[height=358,width=600]

.Syntax

Use OWL::
Because the OWL standard (Web Ontology Language) provides more options
than conventional RDF syntax, this option is always recommended. The
exception is when the RDF file is to be reused by another system that does
not process OWL.

Use KRDF::
The KRDF syntax contains additional elements for representing knowledge
graph-specific features such as:
* Schema of properties with types as domains
* Extensions
+
[NOTE]
You should always enable this option for transfers between knowledge
graphs, but disable it for exchange with external systems.

.Scope

Export schema only::
This function is used to export only the schema of the knowledge graph
without instances, in case of an intended schema transfer.
```

```
Export names as rdfs:label::
Labels are not exported as attributes, but in the form of a label tag with
the syntax `<rdfs:label xml:lang="de">`.

Export meta properties::
According to the official RDF specification, the representation of meta
properties is not intended.
However, meta properties can be interpreted as a construct of a statement
about another statement (_reification_).

Export extensions::
Enables the export of extensions of semantic elements.

Extended comments::
Generates extended XML comments. Comments are added to the exported RDF
file that organize the document by objects (instances), related objects
(relation targets), and referenced schema, and additionally contain
statements about mapped relations with source and target objects.

.IDs

Use IDs (rdf:ID)::
Exports shortened RDF IDs for identifying objects, if possible.

Use URLs (rdf:about)::
Exports complete URLs for identifying objects.

Create attributes for generated URLs and IDs::
The generated RDF IDs/URIs are stored in the _rdf:ID_ or _rdf:about_
attributes.

Do not use stored URLs and IDs::
New RDF IDs/URIs are generated instead of using values stored in the
_rdf:ID_ or _rdf:about_ attributes.

.Frame IDs

Export frame IDs of types and objects::
The frame ID is exported as an additional property `krdf:frameID`.

Export RDF IDs of properties::
Exports a reification ID (_rdf:ID_) for all properties. If this option is
not enabled, a reification ID is exported only when needed.

:leveloffset: 4
```

```
:leveloffset: 3

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#08300f7a-76d7-468e-9b7a-91640b946dda]
= External Index in Elasticsearch

Elasticsearch is an open-source search engine based on Lucene, designed
for indexing, searching and analyzing large volumes of data. Its strength
lies primarily in value and full-text searches. In i-views, it is possible
to export data from the semantic network to Elasticsearch using a Mapping.
This creates an external Index that can be used with numerous search
functions and options. For more information, the official website of
Elasticsearch can be found link:https://www.elastic.co/[here].

== Creating a data source

To utilize the interface from i-views to Elasticsearch, the first step is
```

```
to create a new data mapping. For this purpose, in the working folder,
select the "Elasticsearch" data source using the **New mapping of a data
source** button, and define a name (see 1.5.1.2). After confirmation, the
configuration view will then open:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/1_eng.png[]

The view can be divided into four areas:

. Mapping
. Metadata
. all defined fields (local schema)
. later an overview of all fields in Elasticsearch (schema in
Elasticsearch)

At first, all necessary metadata for the external index should be
specified.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/2_eng.png[]

. (mandatory) The URL on which the external index in Elasticsearch is
hosted
. (optional) A username, if a user is defined
. (optional) A password, if a user is defined
. (mandatory) The preferred name of the external index

After filling out all mandatory fields, the local schema can now be
defined.

== Local schema

The schema, which is initially created locally, determines how the
external index will appear on Elasticsearch. Fields are defined to specify
how the data will be stored. The fields represent the columns of the
external index. For each field, a name and data type must be defined as a
basic requirement. This can be done in the following view:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/3_eng.png[]

. **Add**
```

```
Adds a new field with the specified name and data type.
. **Edit**
Name, data type and modificators can be changed. Analyzers can be set as
modificator for the selected field. These will be explained in a later
section (see 1.5.5.9)
. **Delete**
Deletes all selected fields.
. **Import fields from data source**
Overrides the local schema with the schema in Elasticsearch.
. **Export properties to data source**
Uploads the local schema to Elasticsearch.

The local schema, which includes all fields, defines the structure of the
external index. An example of such a schema is as follows:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/4_eng.png[]

After defining a new schema, the local schema can be exported to
Elasticsearch by using the arrow down **Export properties to data
source**. This opens the view of fields in Elasticsearch where the
transferred schema is displayed:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/5_eng.png[]

. **Refresh (F5)**
The schema is reloaded from Elasticsearch.
. **Reset all fields**
The entire external index is reset. All exported data is deleted. Only the
schema remains intact.

[NOTE]
In general, handling fields requires great care, and it's advisable to
make minimal or, ideally, no changes to them during operation. Modifying
fields while an external index exists could lead to anomalies. Therefore,
significant schema changes should prompt the recreation of the external
index using the eraser **Reset all fields**.

After creating the desired schema, the next step is to define a mapping.

== Mapping

The mapping defines which data will ultimately be exported to the external
```

```
index. To do this, the data source is selected in the mapping section.
Then, step by step, the structure of the mapping can be defined (see
1.5.1.3). An example of a mapping could look like the following:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/6_eng.png[]

If the local schema has already been created, a selection dialog will pop
up for each mapping part, allowing to choose a field:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/7_eng.png[]

Here, the currently created part of the mapping can be assigned to a field
for export. If **- none -** is selected or if there is no local schema
yet, an assignment can also be made later on. For this, a part in the
mapping must be selected, and the desired field must be chosen for **Map
to**:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/21_eng.png[]

After the mapping was defined and assigned to the fields in the local
schema, configuring a unique identification of the objects to be exported
is essential for smooth operation.

== Identification

For a clear identification, Elasticsearch automatically generates unique
IDs for every exported Object. However, it is advisable to use the
existing IDs of the objects in the semantic network to ensure a unique
identification. This establishes a direct connection between the objects
in the semantic network and the entries in the external index, making
future operations much easier.

This can be achieved by reimporting the schema from Elasticsearch after
exporting the local schema. This process reveals a new field named "_id"
in the local schema, which is generated automatically by Elasticsearch in
order to store unique IDs for every entry in the external index.

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/9_eng.png[]
```

```
[NOTE]
The ID field should only be used for mapping unique IDs from the semantic
network. Mapping other types to this field can significantly affect the
functionality of the external index or even lead to a complete loss of
functionality.

Previously, the ID field provided by Elasticsearch was only loaded into
the local schema. However, a new mapping for the ID must be added
specifically for the root object of the mapping:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/10_eng.png[]

The IDs of the objects to be exported from the knowledge network have now
been added to the mapping and assigned to the ID field. Additionally, it
should be ensured that the root objects are also identified by these IDs.
When selecting the root object, in the Identify tab under Identify object
using the following mappings, only the previously added ID mapping should
be defined. If the root object is identified by one or more other
attributes, these should be deleted and only the ID mapping should be
added. It should look like this:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/11_eng.png[]

During an export, the entries in the external index are now identified
through the IDs of the root objects defined in the mapping. This is
crucial for associating objects in the semantic network with the
corresponding entries in the external index and enables additional
functions, such as automatic updating of the external index when data has
changed internally (synchronization).

== Synchronization

In order to keep the external index current and consistant to the semantic
network, before the initial export, automatic synchronization should be
activated under the **Options** tab:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/24_eng.png[]

This generates a new field with the name "_dependantIDs" for the external
```

```
index which is used as an auxiliary index for keeping data current. In
addition to that, a trigger must be configured that activates when data in
the semantic network is changed and also starts the update process:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/25_eng.png[]

Afterwards a configuration view for the trigger will open where the
following settings should be configured:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/26_eng.png[]

. Selecting the mapping on which the trigger should be applied on.
. Selecting the **Primary core element** parameter.
. Activating **Update automatically**.

After the trigger was configured, the external index will now be
automatically updated when internal data is changed accordingly.

== Import and Export

At this point, an import/export can now take place. The buttons for these
actions are located above the mapping section:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/12_eng.png[]

. **Import**
Data is imported from the external index to the semantic network.
. **Interactive import**
. **Export**
Data is exported from the semantic network to the external index.
. **Move up**
Moves up the selected part of the mapping
. **Move down**
Moves down the selected part of the mapping

After exporting data to the external index, the interface between i-views
and Elasticsearch provides a variety of options.

== Browser tool Elasticvue
```

```
The exported data can be viewed using the browser extension Elasticvue.
The extension supports the most popular browsers and can be downloaded
from link:https://elasticvue.com/[elasticvue.com/].

After successful installation, Elasticvue can be opened among the
installed extensions. Upon initial opening, a setup is required where the
address, through which the Elasticsearch cluster is running, is entered
under URI. Optionally, a username and password can also be provided.

[NOTE]
The URI must be identical to the URL entered in the mapping.

After the initial setup, a redirection is performed to the dashboard,
which will now appear directly when reopening the extension. The
navigation bar of the dashboard looks as follows:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/vue_navigation.png[]

Under **default cluster**, all existing clusters can be selected, managed,
and new clusters can be created. Additionally, the address of a cluster
can be viewed here, which must be entered in the metadata of the mapping
in the Knowledge-Builder. Under **Home**, many more metadata about all
clusters are displayed. Each cluster consists of multiple nodes.
Information along with a resource overview for each node can be found
under **Nodes**. Each external index can be distributed across multiple
shards, which is displayed under the **Shards** tab. The most important
tab for viewing the exported data is **Indices**, where all external
indexes are listed. Selecting an index provides a detailed overview of the
exported data:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/vue_übersicht.png[]

Each row shown represents an exported object from the semantic network in
i-views. Since Elasticsearch stores all data in JSON, individual records
can be displayed as JSON under **Show** or all data can be downloaded in
this format using the **Download as JSON** button. The overview also
provides the option to search the data using the search field **Search**,
which is based on a simplified search syntax. For more complex searches, a
more intricate query can be defined under **Custom Search** using JSON.

Comprehensive documentation for everything related to Elasticsearch,
particularly helpful for custom searches, is available on the
```

```
Elasticsearch website at
link:https://elastic.co/guide/index.html[https://elastic.co/guide/index.ht
ml].

== Queries and facets

To make effective use of the external index, Elasticsearch queries need to
be configured. Using these queries, Elasticsearch's search functions and
options can be employed to search through the exported data and retrieve
relevant results. To configure such a query, it must be created in the
working folder. A configuration view will then appear:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/13_eng.png[]

The configuration view is divided into four sections:

**1. Data source**
Here it is defined for which mapping the query should be provided. Again,
the mapping must be registered first before it can be selected.

**2. Parameters**
In this section, the parameters to be used for the query are defined.
Additionally, the search logic, which plays a crucial role in the query,
is established here. To define logic for the query, search criteria must
be added:

//TODO: Screenshot neu, der Dialog sieht anders aus
image::/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/14_eng.png[]

The selection offers the following options:

General parameters::
Combination::: Combines individual search criteria into a consolidated
logic.
Field parameters::
Range::: A match occurs when the value of a field falls within the
specified range.
Existance::: A match occurs when the value of a field exists.
Term::: A match occurs when the value of a field is identical to the given
value.
Fuzzy::: A match occurs when an indexed expression is similar to the
provided expression (tolerant to swapped, modified, missing, and extra
```

```
characters).
Fulltext::: A match occurs when the specified expression is found in
another using full-text search.
Geometry parameters:: These parameters are only applicable to fields of
type _shape_ or _geo_shape_.
Distance::: A match occurs if the parameterized reference point is within
a defined distance from the geometry of the field. The distance can be
specified with a unit (e.g., `km`, `yards`, `nmi`, ...). If no unit is
given, "meters" are assumed.
+
[NOTE]
====
Distance queries are only applicable to fields of type _geo_shape_.
====
Bounding Box::: A match occurs if the geometry of the field is within the
specified bounding box.
+
[NOTE]
====
Elasticsearch uses the notation `BBOX(<minLon>, <maxLon>, <maxLat>,
<minLat>)` for the Bounding Box parameter. E.g. for a box with the corner
points 48° N 8° E and 50° N 10° E, the resulting parameter value is
`BBOX(8,10,50,48)`.

Bounding Box queries are only applicable to fields of type _geo_shape_.
====
Contains::: A match occurs if the geometry of the parameter is completely
contained in the geometry of the field.
Disjoint::: A match occurs if the geometry of the parameter does not
overlap or touch the geometries of the field.
Within::: A match occurs if the geometry of the parameter completely
contains the geometry of the field.
Intersects::: A match occurs if there is any overlap between the geometry
of the parameter and the geometry of the field.

Depending on the selected search criterion, there are various settings in
the configuration view. With a few exceptions, these are similar, so the
key settings can be demonstrated using the example of the range criterion:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/15_eng.png[]

. Here, the field of the external index on which the search criterion is
applied is specified.
. In general, this is where parameters for a query can be specified. In
```

```
the case of the range criterion, for example, there are two parameters
that form the upper and lower bounds, and the option to change the
comparison operators.
. Here, the logic can be selected based on which the query operates. The
following options are provided:
must:: Equivalent to logical AND. A match ocurs when all search criteria
are satisfied. Increases the Elasticsearch score for a match.
should:: Equivalent to logical OR. A match occurs when at least one
criterion is satisfied. Increases the Elasticsearch score for a match.
must not:: Opposite of must: A match occurs when the specified value does
not satisfy the search criterion.
Filter:: Equivalent to must, but the Elasticsearch score remains
unaffected.
. If enabled: Checks whether the specified parameters have already been
defined in the system. If the parameters do not exist yet, the query will
not be executed.
. Returns an empty set if the value does not exist in any entry.
. Can be used to weight individual search criteria. The entered value
(float) serves as a factor that is applied to matches when calculating the
Elasticsearch score. The default value is 1.0. The higher the value, the
more weight the search criterion carries in the Elasticsearch score
calculation.

Exceptions with additional specific configuration options include:

**Term:**

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/16_eng.png[]

. Here, a type for the search criterion _Term_ can be selected. Possible
types are:
Prefix:: A match occurs when the specified search term matches the
beginning of an indexed expression (Example: Search term "Te" -> Match
with "Term").
Wildcards:: Wildcards can be used in the search, replacing any parts of a
term (Example: Search term "Wild*" -> Match with "Wildcards" | The
asterisk serves as a placeholder for any number of letters).
Regular Expression:: Makes use of regular expressions for searching.
Equal:: A match occurs only if the specified search term is identical to
the indexed expression.
. If disabled: A search term is considered identical to an indexed
expression even if the capitalization of any letters is different.

**Full text:**
```

```
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/17_eng.png[]

. Similar to the search criterion "Term", a type can also be selected
here:
Matching:: A match occurs when the specified expression is identical to
the indexed full text.
Contains phrase:: A match occurs when the specified expression matches a
part of the indexed full text.
Begins with phrase:: A match occurs when the indexed full text begins with
the specified expression.
Apply query syntax:: A pre-defined search syntax by Elasticsearch is
applied.
Simple query syntax:: A pre-defined simplified search syntax by
Elasticsearch is applied. It offers fewer options than the normal search
syntax but is more error-tolerant.

**3. Fields**
After selecting a mapping, the fields of the external index are displayed
here. Additional functions and options can be assigned to each field
individually. For this, there is the following configuration view:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/18_eng.png[]

The fields of the locally created schema are visible here. If a specific
field is specified in a query to be searched with a custom parameter, the
name of the parameter is displayed in the corresponding row under
**Parameters**. For each field, the **Cause** can be activated. If this is
the case, the values of this field will be displayed in the result table.
**Highlight** can be activated for each field to highlight matches on this
field in the result table. Additionally, a facet can be added to each
field, which is done by selecting the field and using the **Add Facet**
button. Facets can also be removed with **Remove Facet**. Facets provide
the ability to group and categorize matches using terms or values. The
Refresh button reloads the view.

[NOTE]
Facets are not compatible with every data type, such as Text or Search-as-
you-type, but only for groupable values like integers and floats,
keywords, or time and date values.

**4. Settings**
```

```
Additionally, further settings can be made that affect the result set:
image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/19_eng.png[]

. **Restrict result set size** If enabled: The results of a query to the
external index can be limited by inserting a limit (integer) into the
input field. All results are based on a score assigned by Elasticsearch,
where a higher score indicates greater relevance. Limiting involves
listing the most relevant results in descending order.
. **Minimal Elasticsearch Score**
If enabled: The results of a query are limited based on the score assigned
by Elasticsearch. The value entered into the input field (float) serves as
the threshold. Only results with a score above this value are considered.

To test the search behavior of a query, the testing environment can be
opened to the right of the settings:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/22_eng.png[]

. Here, a search text can be entered.
. If parameters are defined during the configuration of queries, they are
listed here. It is possible to set a custom value for each parameter
individually using **Set value** for the test search. The value can also
be cleared with **Reset**. Once all values are set or a search text is
entered, the test search can be initiated by clicking **Search**.
. After executing the test search, all matches of the query are listed
here. The Reason column indicates for each match why the object is
considered a match with respect to the query. In the Quality column, each
match is assigned a score. This is displayed as a float, where a value of
1.0 represents the highest score. The score indicates how precisely the
search criterion matches the result compared to all other matches. The
higher the score, the more match and relevance.

[NOTE]
It is advisable to refrain from setting values for the parameters and
using the search text simultaneously, as unexpected effects may occur.

== Settings

In addition to exporting data, metadata for the external index can also be
exported. These can be displayed using Elasticvue:

image:/builds/i-views/documentation/documentation-
```

```
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/vue_meta.png[]

After expanding the gear icon, metadata can be viewed under **Show info**.
Among the metadata are details such as the name of the external index,
possible aliases, the exported schema with all defined fields, as well as
settings. Some settings are automatically generated during the creation of
the external index. However, there are also settings that can be manually
configured, including analyzers, for example. More information can be
found
link:https://www.elastic.co/guide/en/elasticsearch/reference/current/index
-modules.html[here].

In the Knowledge-Builder, the settings can be configured in the
Elasticsearch mapping under the **Settings** tab. The input field can be
used to define it in JSON:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/20_eng.png[]

. The input field for the settings. The definition of the settings is done
in JSON.
. This allows inserting a template into the input field, serving as a
basis.
. The settings from Elasticsearch are loaded into the local input field.
. The defined settings are sent from the input field to the external
index.

[NOTE]
Both successful setup of the settings and an error display a status under
the input field. In case of an error, the type of error is also indicated.
This could be, for example, an invalid format or a connection error to the
external index.

=== Analyzer

Another feature of Elasticsearch is the so-called analyzers. Elasticsearch
provides a variety of text analysis functions that can be formulated in
JSON and bound to a field of the external index as an analyzer. This
allows data for individual fields to be adjusted during export and expands
search capabilities. Detailed documentation on Elasticsearch can be found
link:https://elastic.co/guide/en/elasticsearch/reference/current/analysis.
html[here].

In the Knowledge-Builder, there is a configuration view similar to the one
```

```
for the settings when selecting the **Analyzer** tab. After defining an
analyzer und exporting it to the external index, it can then be selected
by editing a field:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/external-index-in-elasticsearch/gfx/23_eng.png[]

Next to a few basic preset analyzers, the custom analyzer is shown at the
top of the list. If more than one custom analyzer is defined, it will also
be shown at the top. After choosing an analyzer, it will now be displayed
under **Modifiers** in the local schema.

[NOTE]
Analyzers are only compatible with text fields!

After an analyzer was bound to a field, it will be applied to the external
index which extends the search options for that specific field.

:leveloffset: 4
:leveloffset: 3

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1
```

```
:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#9a268883-71b2-4ce0-9139-ac58cf223ea1]
= Restore deleted individuals from a back up

The RDF export and import is suitable for restoring deleted individuals
from a backup Knowledge Graph. Proceed as follows to do so:

. Open the backup Knowledge Graph in the Knowledge Builder
. Create a new folder and save the individuals to be restored to it. To do
so, right-click to open the context menu in the list view of the
individuals to be copied, and select "Copy content to new folder" while
selecting the new folder as the destination.
. Open the RDF export on the newly created folder using the context menu
. Specify a file name in the export dialog, select the options "Use URLS
(rdf:about)" and "Use frame URLs (krdfframe:)" and execute the export:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/restore-deleted-individuals-from-a-back-
up/gfx/222_RestoreIndividualsRDFExport.png[height=358,width=600]

[NOTE]
The option "Use KRDF" results in i-views additionally copying specific
content that cannot be mapped in full by means of RDF syntax.

. Close the Knowledge Builder and open the target graph in the Knowledge
Builder
. Open the RDF import dialog in the main menu under Tools > RDF > RDF
import:

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/restore-deleted-individuals-from-a-back-
up/gfx/223_RestoreIndividualsRDFImportMenu.png[height=213,width=451]

. Select the file and press "Next":

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/restore-deleted-individuals-from-a-back-
up/gfx/224_RestoreIndividualsRDFImportFileSel.png[height=190,width=500]
```

```
. Deactivate the option "Allow changes to the schema" in the selection
dialog, and activate "Create folder with imported objects":

image:/builds/i-views/documentation/documentation-
asciidoc/build/pdf/users-manual-en/knowledge-builder/import-and-
export/restore-deleted-individuals-from-a-back-
up/gfx/225_RestoreIndividualsRDFImportMapping.png[height=659,width=766]

. Execute import
. Check the restored individuals

:leveloffset: 4
:leveloffset: 3

// Include language-specific variables
// language specific definitions (EN)

:note-caption: Note
:warning-caption: Warning
:caution-caption: Caution

// Language independent definitions
// Style settings
:table-grid: rows
:table-frame: none

// Table layout definitions
// Include with
//   [{name-of-variable}]
//   |===
//   ....
//   |===
:table-name-text: %header, cols="1a,2a"
:table-name-text-extra: %header, cols="1a,2a,1a"
:table-name-text-no-header: cols="1a,2a"
:bridge-port: 8815
:mediator-port: 30069
:iviews-version: 6.1

:inline-button: fit=line

:js-api-url: https://documentation.i-views.com/{iviews-
version}/javascript-api

[#7d38ecd7-0298-4f8f-8cb2-7d4897cb8b77]
```

```
= Transport selected schema

The Admin tool can be used to transfer the entire schema from one
Knowledge Graph to another via RDF export and import. However, if you only
want to transfer selected types, you should consider using the "Copy
schema to folder" function, which is available for all types via the
context menu. This function creates a reference to the selected type
together with all other (property) types that are required to create the
selected type or objects of this type in the target graph.

Once you have collected all required information in a folder, you can
export this and import it into the target Knowledge Graph in the same way
as described in the previous chapter. However, the "Allow changes to
schema" option should be deactivated in this case.

:leveloffset: 4
```