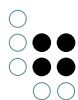


Users' Manual 5.4



Contents

1 Knowledge-BUILDER	3
1.1 Basics	3
1.1.1 The Knowledge Builder application	3
1.1.2 Building blocks	6
1.1.3 Type hierarchy - Inheritance	9
1.1.4 Create and edit objects	11
1.1.5 Graph editor	15
1.2 Definition of schema / model	25
1.2.1 Define types	25
1.2.2 Relation types and attribute types	34
1.2.3 Model changes	45
1.2.4 Representation of schema in the graph editor	49
1.2.5 Metamodeling and advanced constructs	52
1.2.6 Indexing	61
1.3 Searches/Queries	71
1.3.1 Structured queries	72
1.3.2 Simple Search / Fulltext search	91
1.3.3 Search pipeline	96
1.3.4 Model "hit"	112
1.3.5 The search in the Knowledge Builder	114
1.3.6 Special cases	114
1.4 Folder and registration	118
1.5 Import and export	119
1.5.1 Mapping of data sources	120
1.5.2 Attribute types and formats	161
1.5.3 Configuration of the export	163
1.5.4 RDF-import and -export	165
1.5.5 Restore deleted individuals from a back up	175
1.5.6 Transport selected schema	177



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

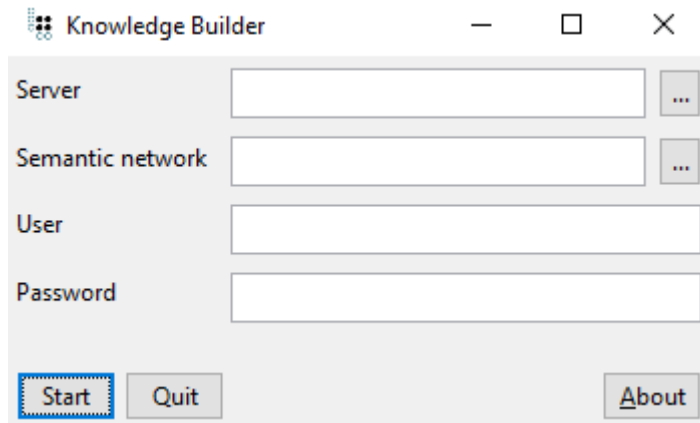
The executable application "kb" is an acronym for the i-views "Knowledge Builder" by which we administer the Knowledge Graph. When talking about the Knowledge Builder, we use special terms for orientation:

- **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) which is run by the Knowledge Builder application.
- **Volume:** The volume comprises all file data of the Knowledge Graph which is accessed by the Knowledge Builder.
- **Knowledge Graph:** The KNOWLEDGE GRAPH is the data core of the i-views Knowledge Graph platform. It is kept as a top type besides the FOLDER section and the TECHNICAL part of the Knowledge Builder application.
- **Element / Semantic element:** An 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. Everything within the Knowledge Builder is built up in forms of this semantic elements logic.



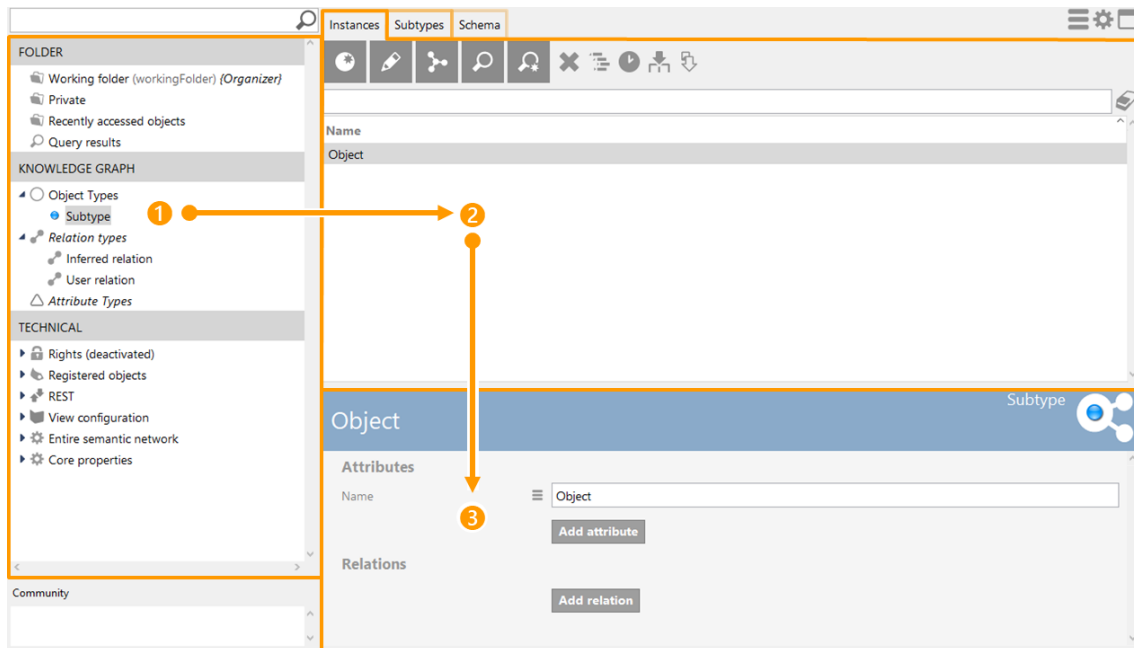
For further information, see the glossary of this documentation.

When we start the Knowledge Buidler 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 preferrably 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-seperated in forms of *serveraddress:portnumber*.
- **Semantic network:** The name of the relevant existing volume must be specified here.
Note: For creating a new volume, the admin tool is needed.
- **User:** User name for volume access.
- **Password:** Password for volume access.

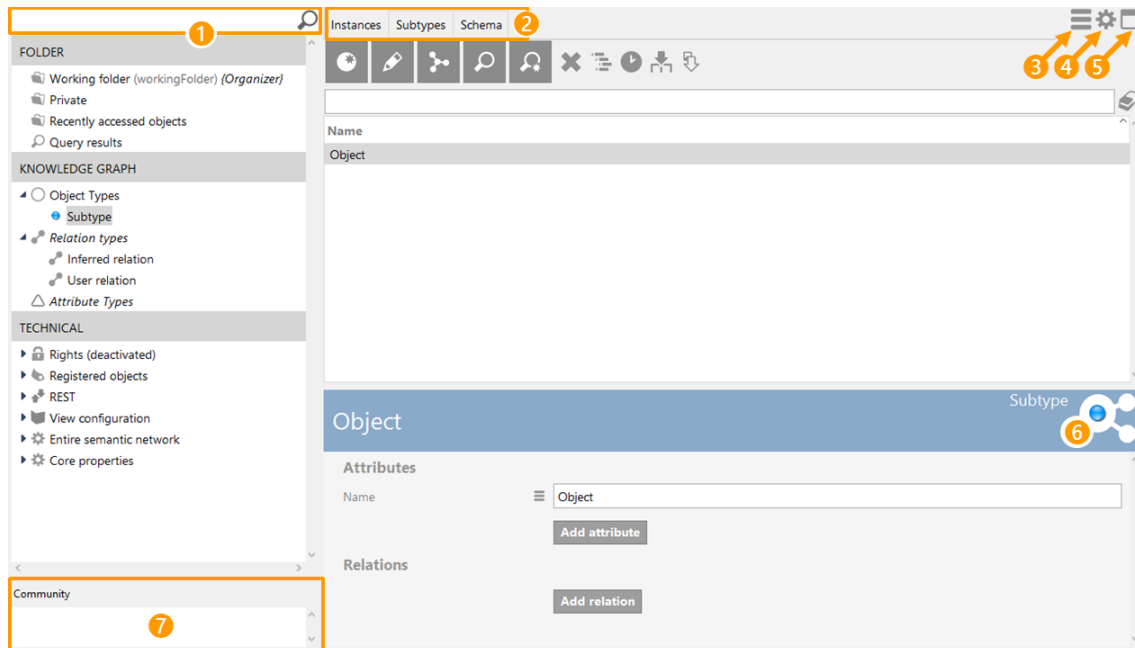
The Knowledge Builder user interface is divided into follwing areas:



- **Organizer:** Type hierarchy view on the left side of the Knowledge Builder screen.
- **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 several table views are defined for one type, they are separated by tabs.
- **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 , then selecting the instance of the list view and by editing the properties in the detail editor .

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



- **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.
- **List tabs:** The list views are divided 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.
- **Global actions:** The global context 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.
- **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.
- **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.
- **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.
- **Community:** If several users are logged in, they are listed here and can be contacted via chat for collaborative work.

For further information, see the following chapters.

1.1.2 Building blocks

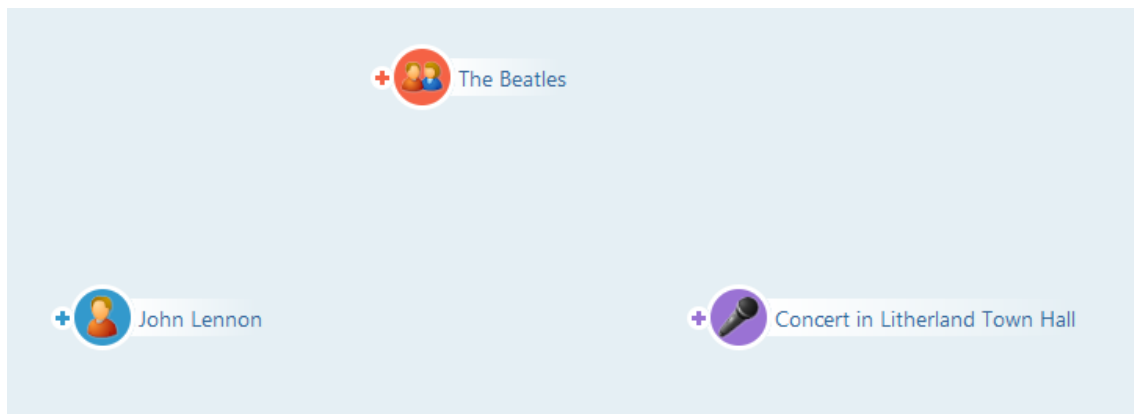
The basic components of modelling within i-views are instances and their types:

- objects

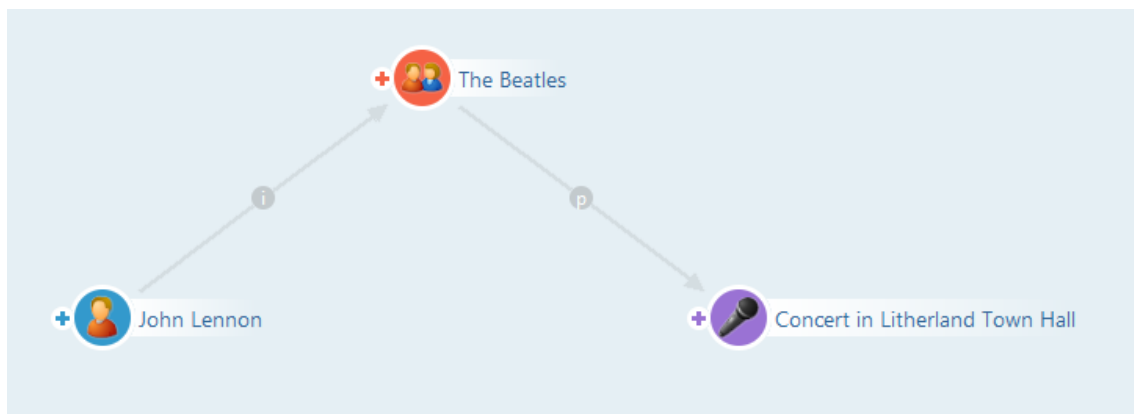


- relations
- attributes
- object types
- relation types
- attribute types

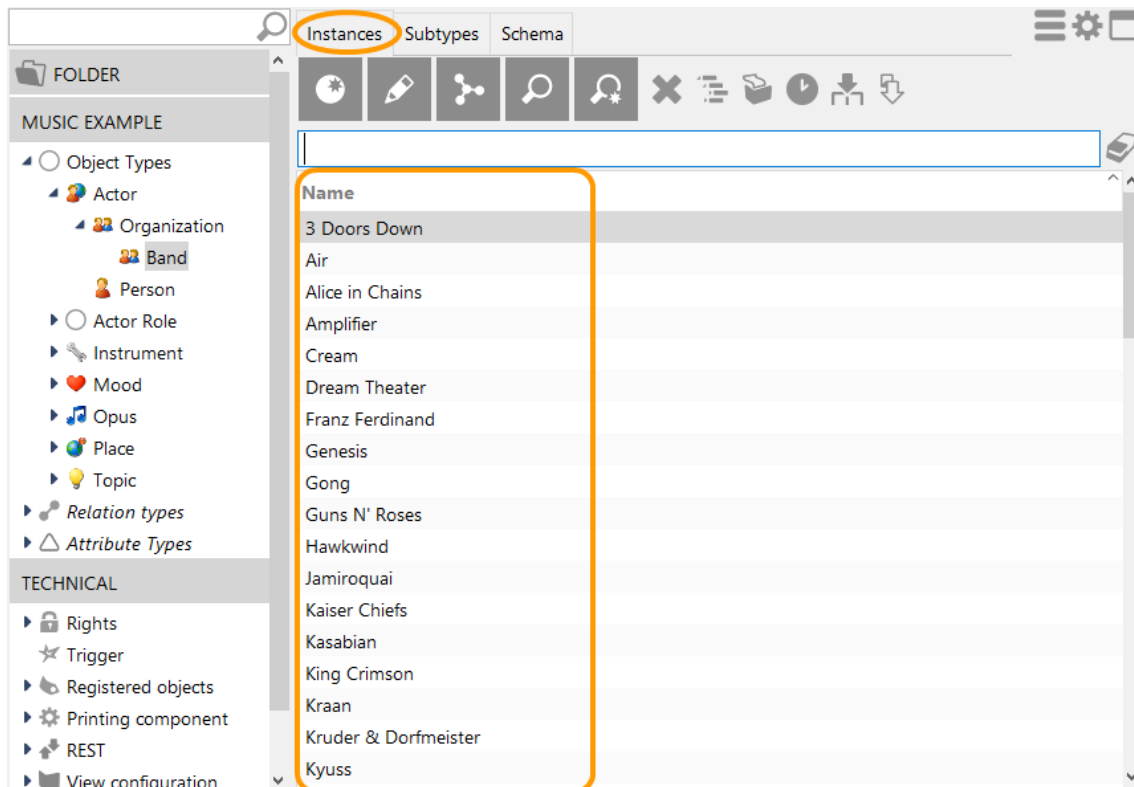
Examples for specific objects are John Lennon, the Beatles, Liverpool, the concert in Litherland Town Hall, the football world cup in Mexico in 1970, the leaning tower of Pisa, etc.:



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".

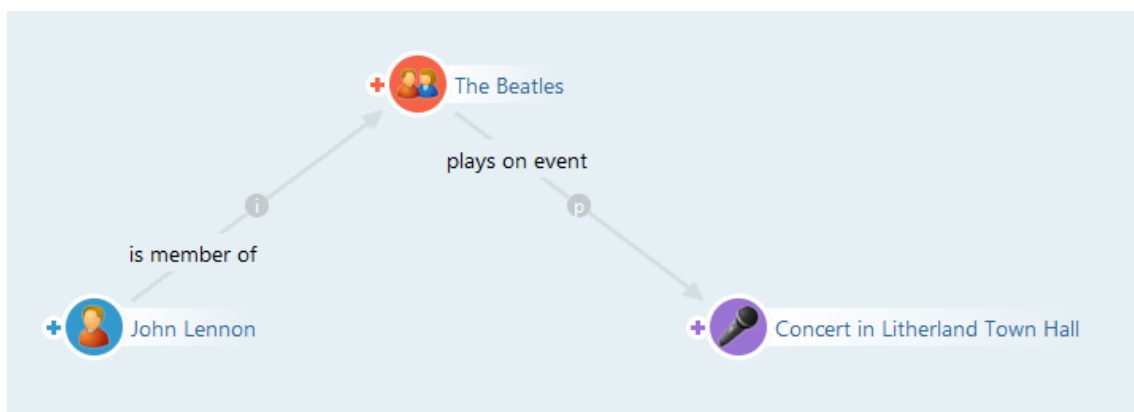


Additionally, we have introduced four types here: specific objects always have a type, e.g. the type of persons, type of the cities, the events or the bands - types which you may freely define in your data model.



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.

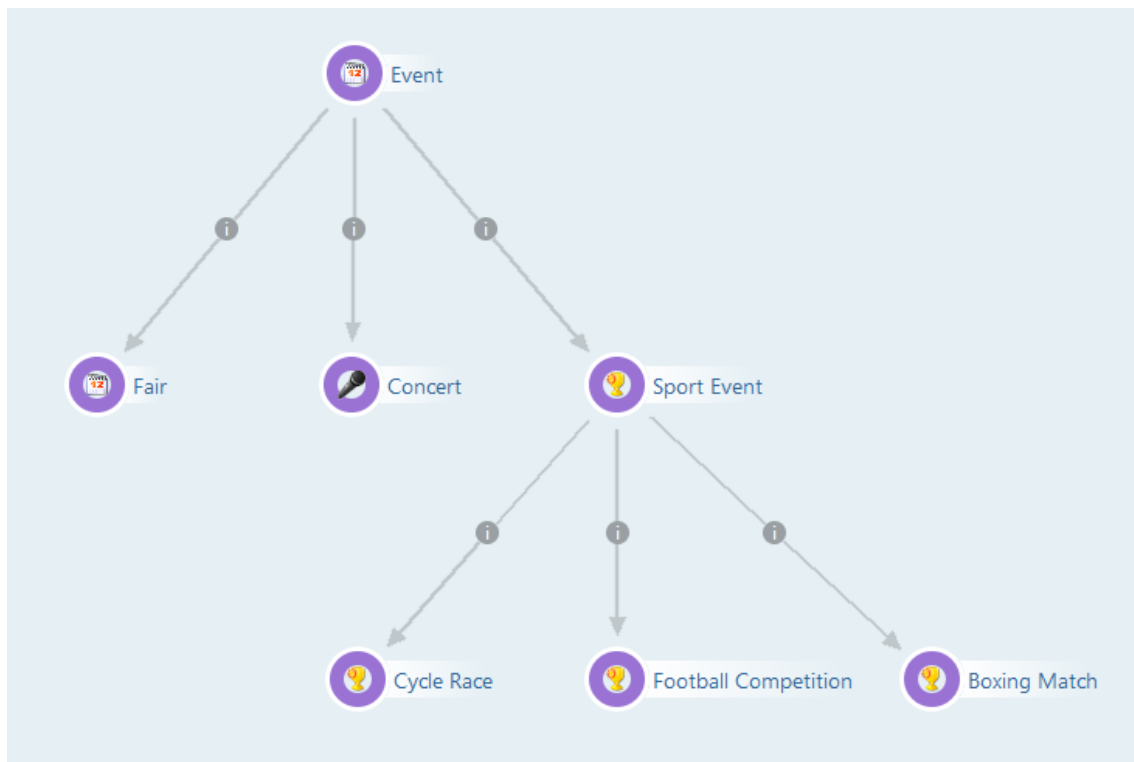


The same applies for attributes: in the case of a person these may be the name or the date of birth. Specific persons (objects of the type 'person') may then have name, date of birth, place of birth, address, colour of eyes, etc. Events may have a location and a time span. Attributes and relations are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

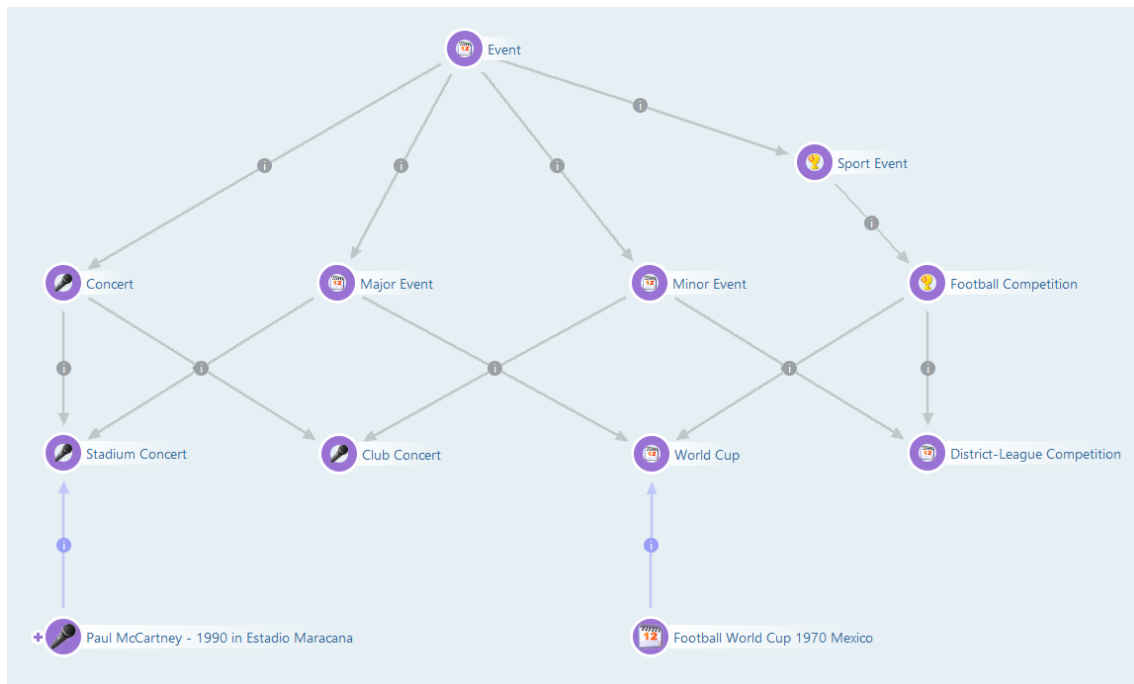
In this manner we obtain a hierarchy of supertypes and subtypes:



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.

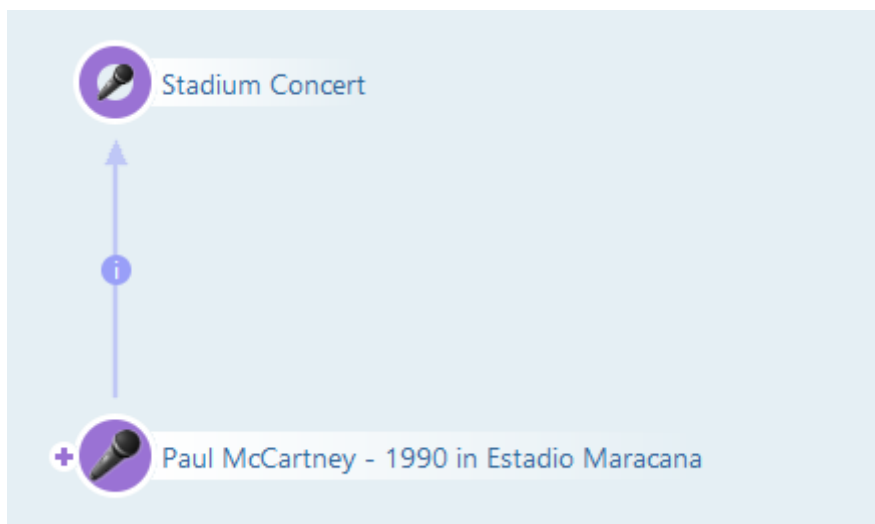
The hierarchical structure does not necessarily have to have the structure of a tree - a type of object may also have several upper types. However, an object may only have one type of object.

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:

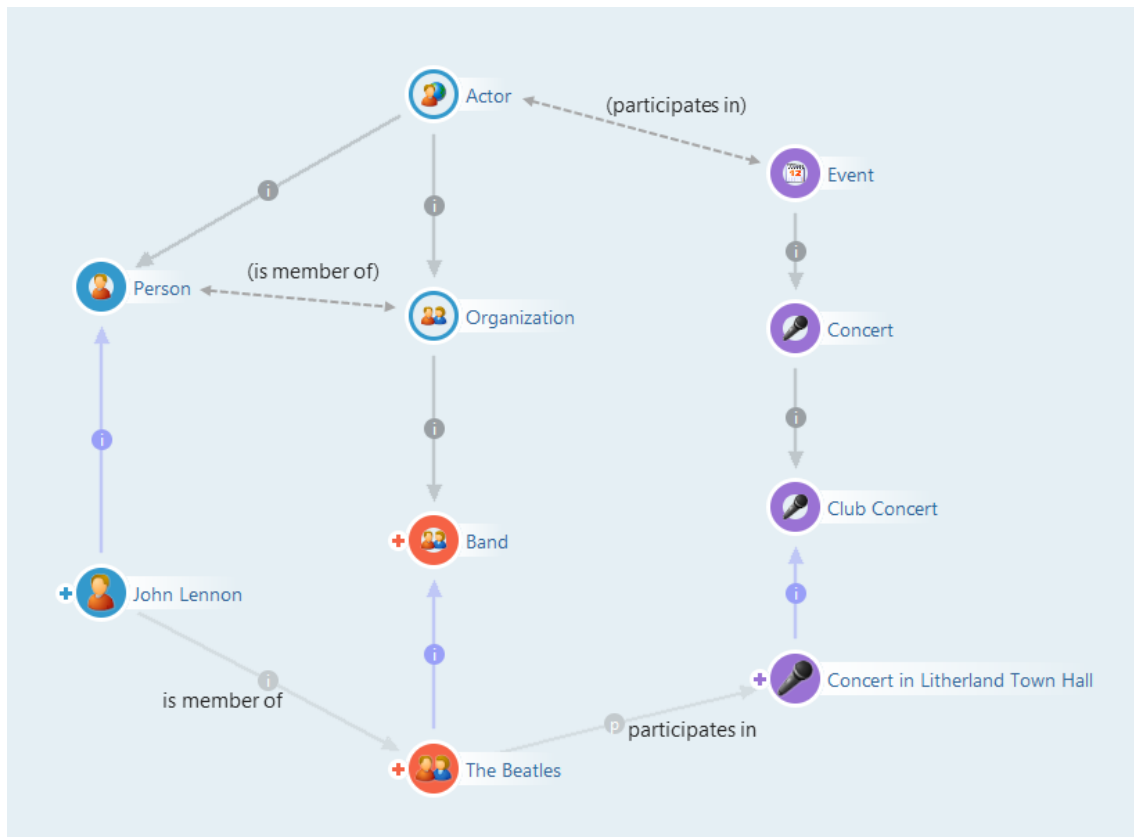


Type hierarchy with multiple inheritance

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.

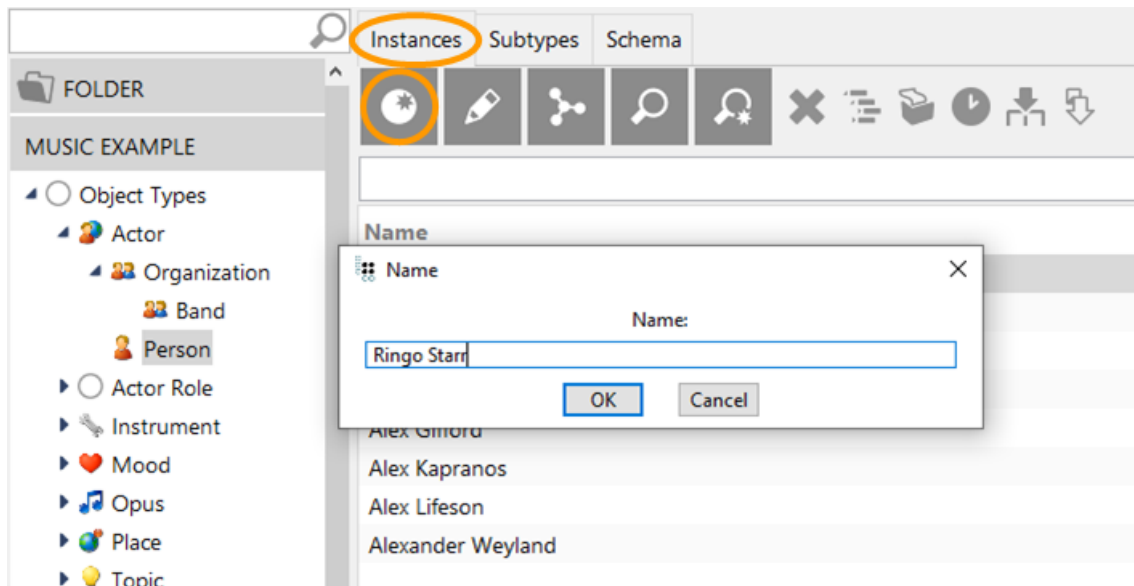


Inheritance makes it possible to define types of relations (and types of attributes) further up in the hierarchy of the object type and hence use them for different types of objects (e.g. for bands and other organisations).

1.1.4 Create and edit objects

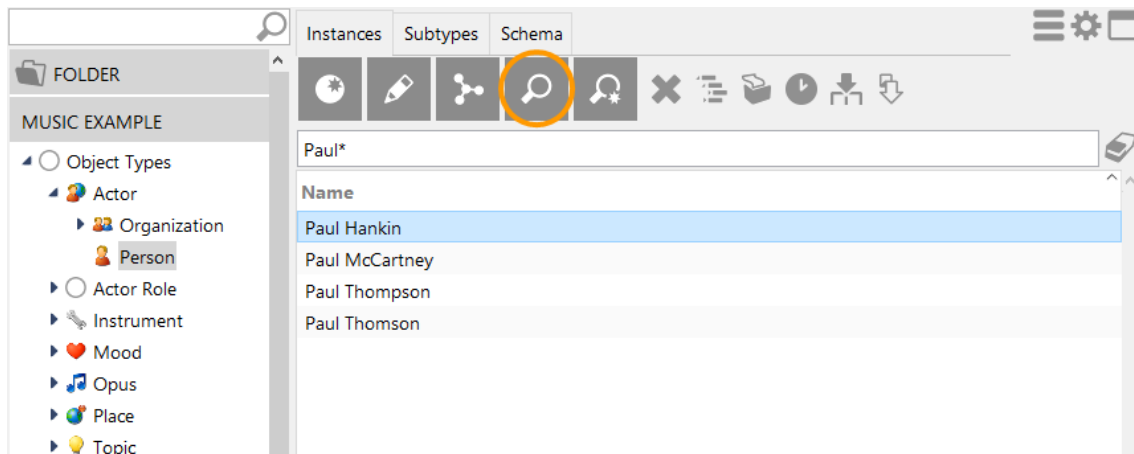
Creating specific objects

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.



Ringo Starr

Person 

Attributes

▶ Name

≡

Ringo Starr

Add attribute

Relations

Add relation

Extensions

Add extension

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,



The Beatles

Band

Attributes

Name

The Beatles

Add attribute

Relations

Is Performer Of

Abbey Road

Has Member

John Lennon

has Place


Liverpool

Has Member

Paul McCartney

Has Member

Add relation

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

Person

Ringo*

Name

Ringo Starr

Name

Ringo Starr

Is Instrumental Musician On

Come Together

Plays Instrument

Drums

Extension

Musician

1 Entry



OK


Create new

Cancel



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.

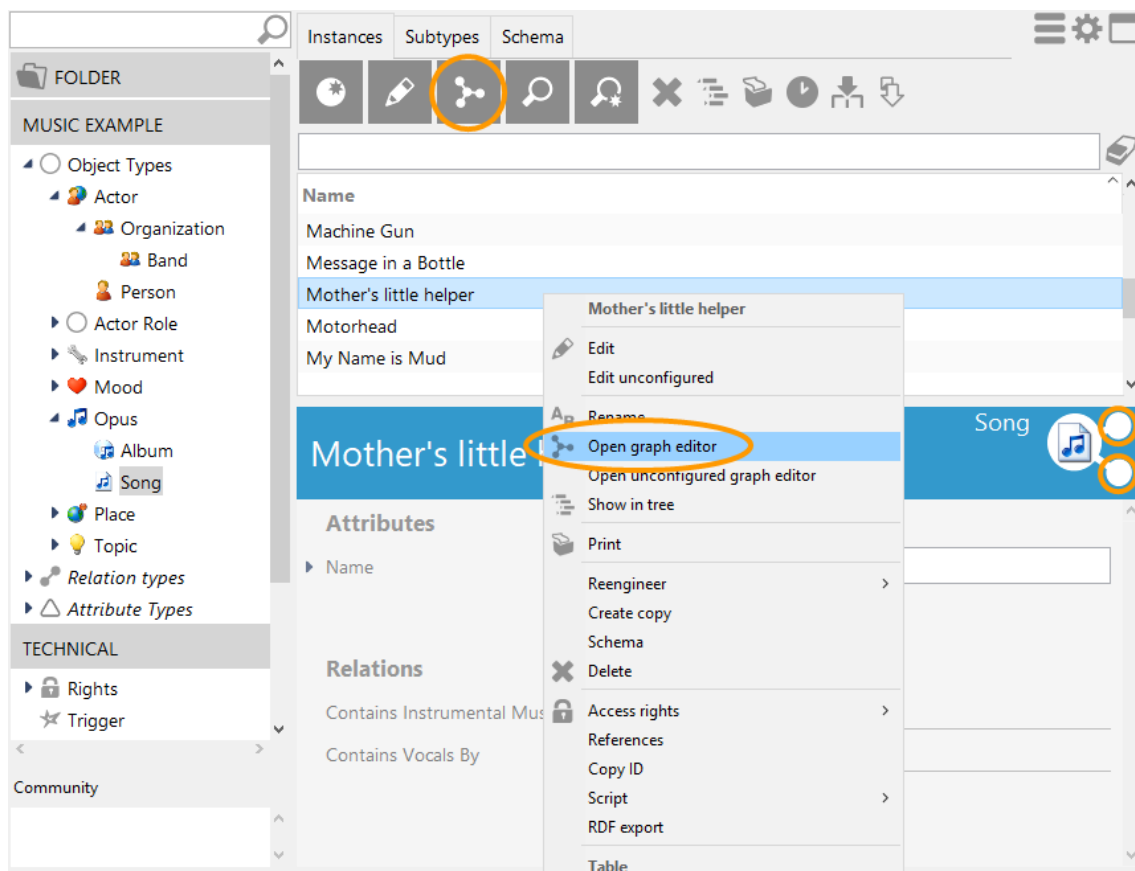
The target object of the relation itself will not be deleted as a result of this however. If an object has to be deleted this is done via the button  in the main window or via the context menu directly on this object.

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

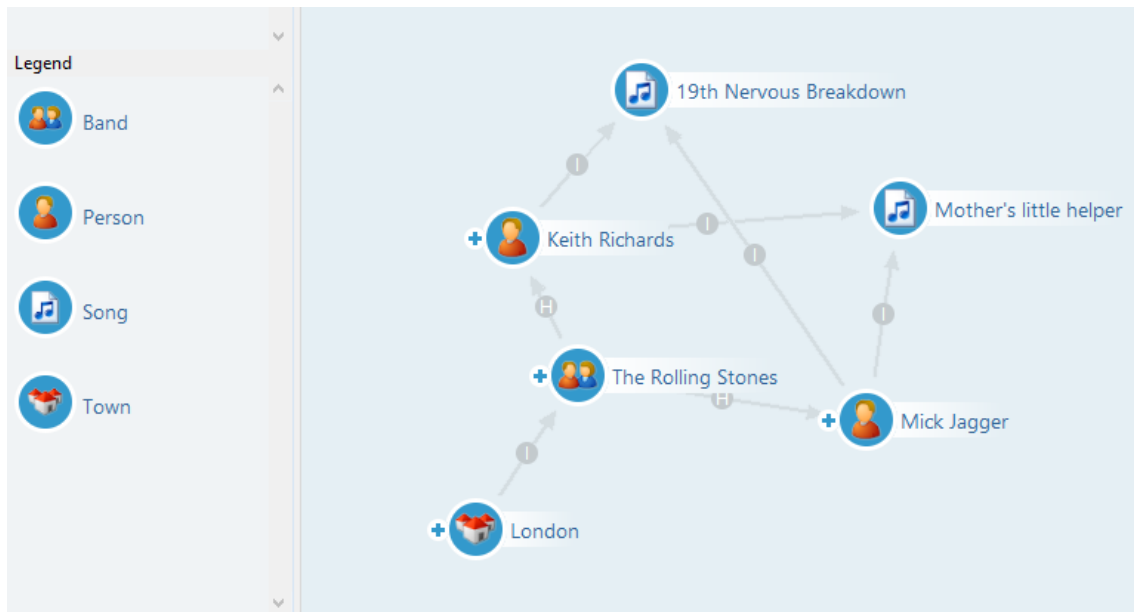
1.1.5 Graph editor

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:



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.



In the graph editor not only a section of the Knowledge Graph may be displayed: objects and relations may be edited as well.

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.

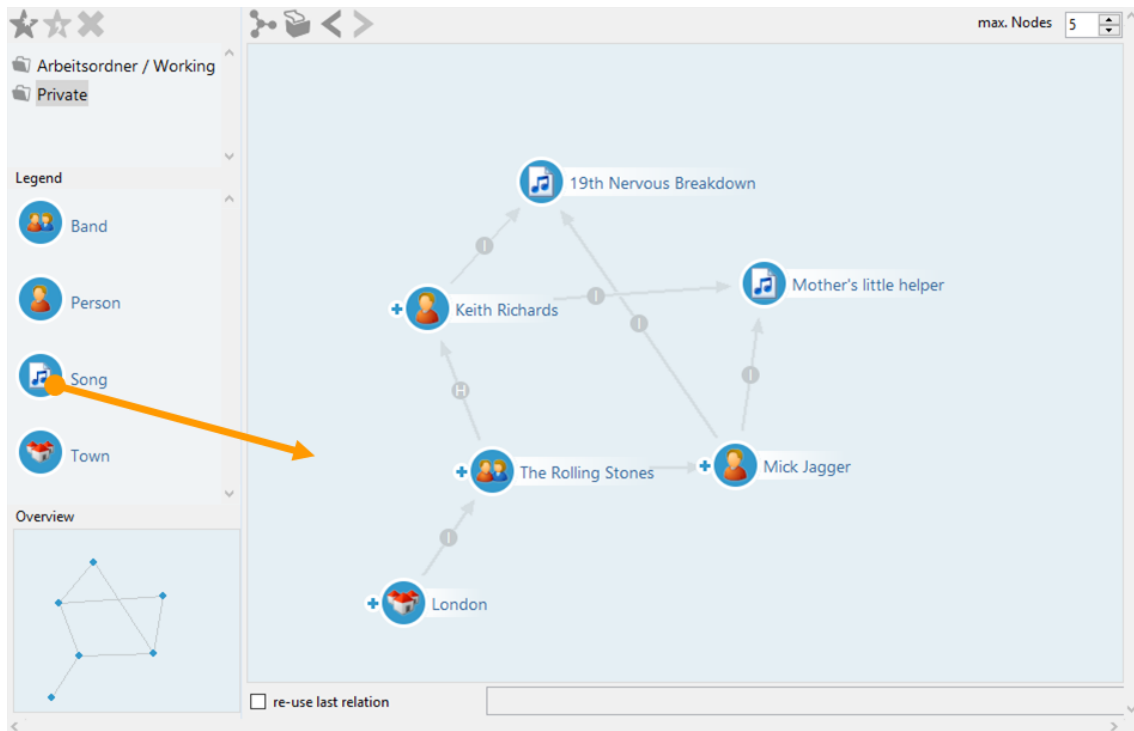


In order to display objects in the graph editor there are different options:

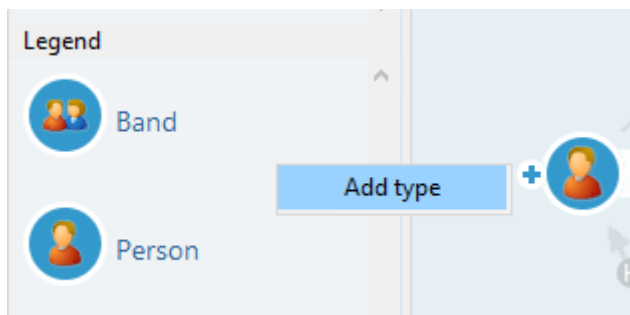
- Objects may be dragged from the hit list in the main window to the graph editor window using drag & drop.
- If the name of the object is known it can be selected via the context menu using the function "show individual".

Shortcut: 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:



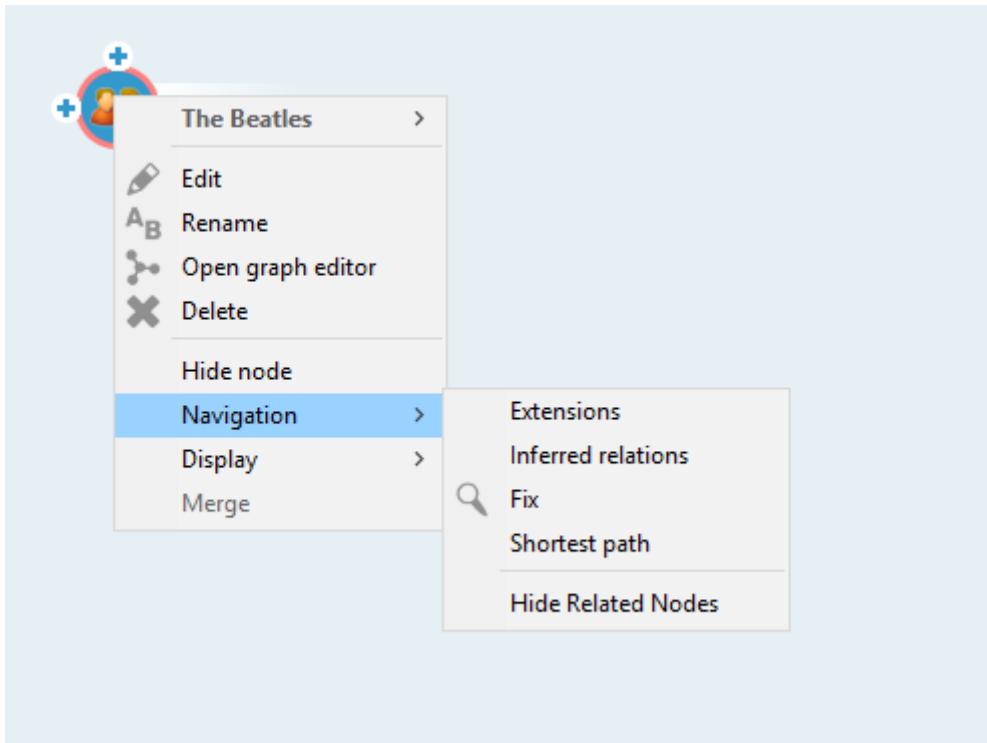
If there are no types of objects to be seen in the legend you can search for them using a right mouse click in the legend area. Following this, the name of the object will be given.



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 name can be changed later on in the Admin tool or the Knowledge Builder. The user created in this way automatically has graph administrator rights. Right-clicking the object in the context menu allows other operations to be executed. For the most part, this context menu provides the same functions as the form editor, however also includes other graph editor-specific components.

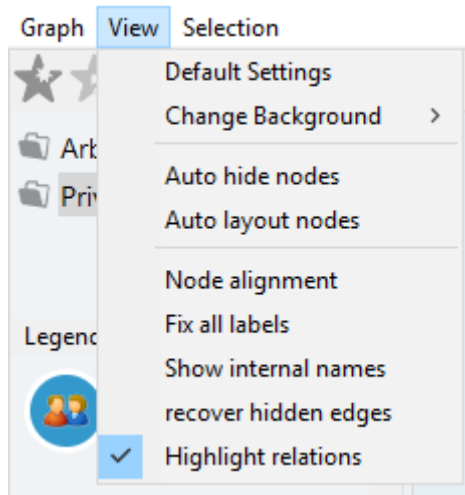



The following graph editor-specific functions are available in this context menu:

- **Hide node:** The node can be hidden here.
- **Navigation - Extensions:** Opens the extensions for an object.
- **Navigation - Calculated relations:** Opens the calculated relations for an object.
- **Navigation - Fix:** Fixes the position of a node in the graph editor, so that it is not repositioned even when the layout is restructured. The fixed node can be undone using the *Release* option.
- **Navigation - Shortest path**

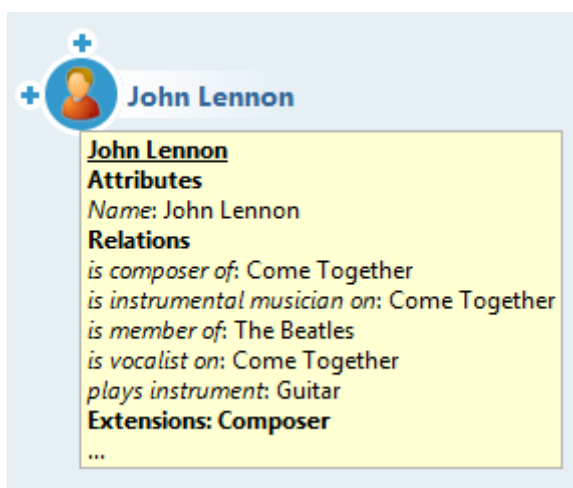
1.1.5.3 View

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



Default settings: Opens the menu with the default settings for the graph editor. This menu is also available in: global setting window  -> register card "personal" -> graph. There you can set whether attributes, relations and enhancements should appear in a small mouse-over-window above the object and how many nodes at a maximum will be visible in one step:

- **Show bubble help with details:** if the mouse pointer stops on one node the details of the first ten attributes and relations will be displayed in a yellow window if bubble help was previously activated. (check "show bubble help with details" in the global setting window register card "personal" graph)



- **Max nodes:** if a node/object has a lot of adjacent objects it often doesn't make sense to show them all by clicking on the drag point.

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:



max. Nodes 5

Auto layout nodes: automatically implements the layout function for newly displayed nodes.

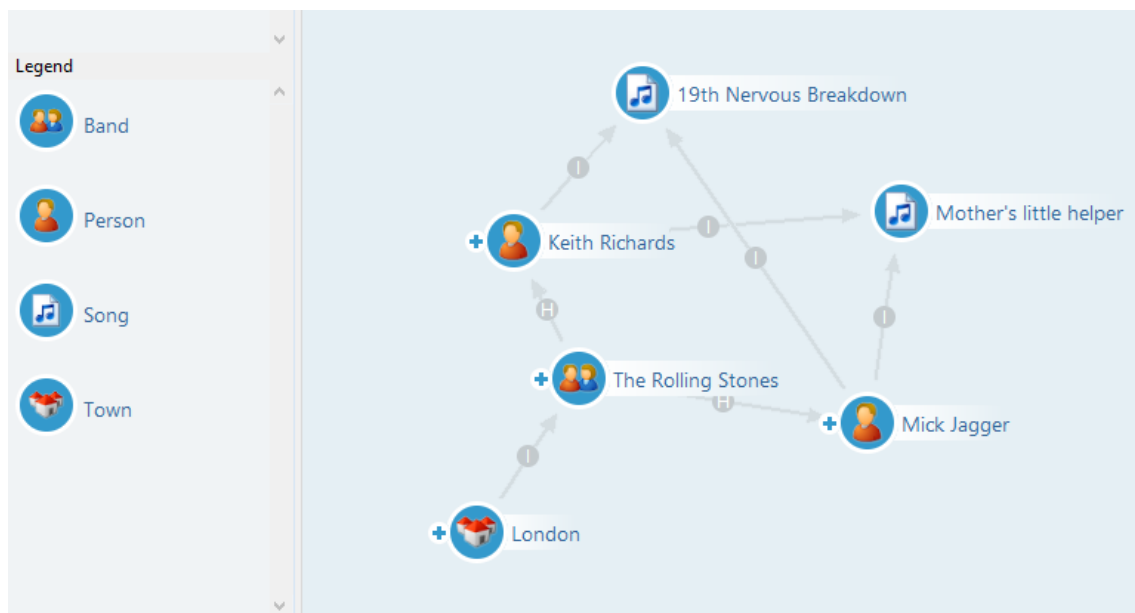
Fix all labels: using this option the names of all relations are always visible, not only when rolled over with the mouse. Alternatively, the description may be fixed directly in the context menu of a relation.

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

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.

By dragging & dropping an entry from the legend into the drawing area you can add or create a new specific object of the corresponding type.

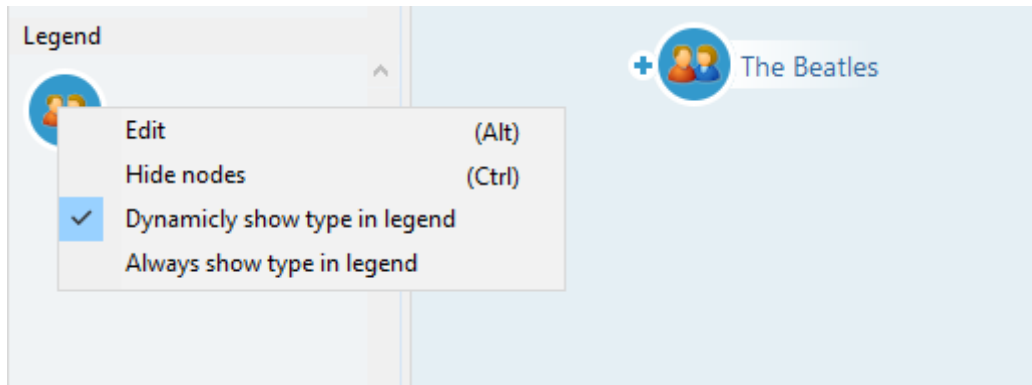
When right-clicking into the legend area, further types can be added permanently to the legend so that objects of that type can be added to the graph by means of drag & drop.

Shortcut: 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 (regard-

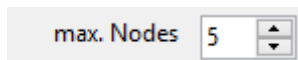


less of whether specific objects of this kind are represented in the image).

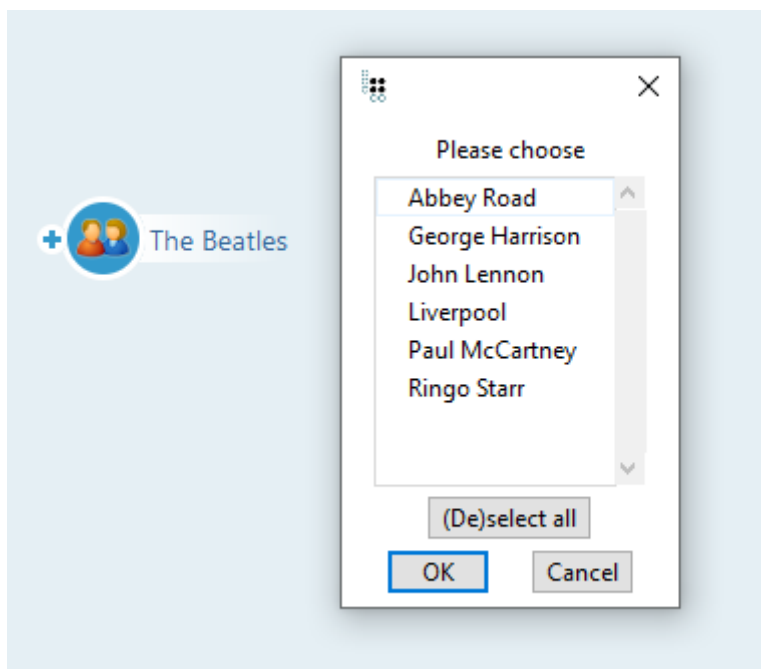


Max. new nodes: If a node / an object has many adjacent objects, it often doesn't make much sense to display all of them when clicking on the drag point. For this reason, the maximum amount of nodes to be displayed at once can be set.

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.



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

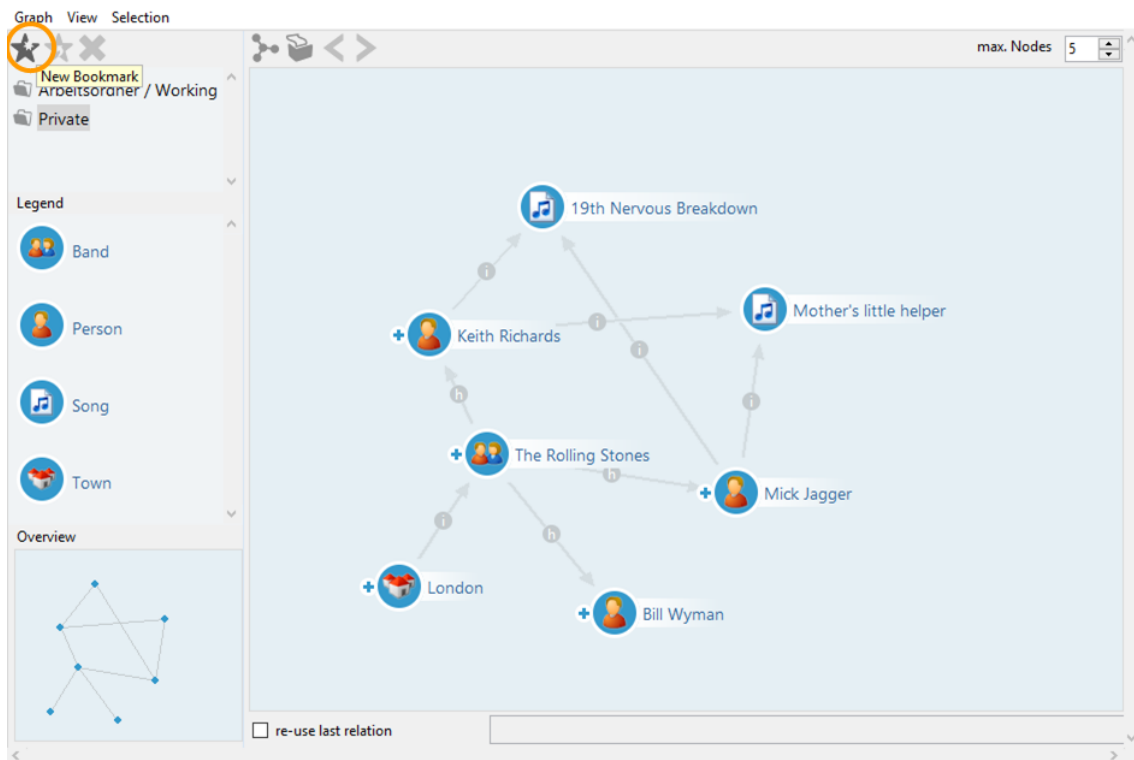




1.1.5.4 Bookmarks and history

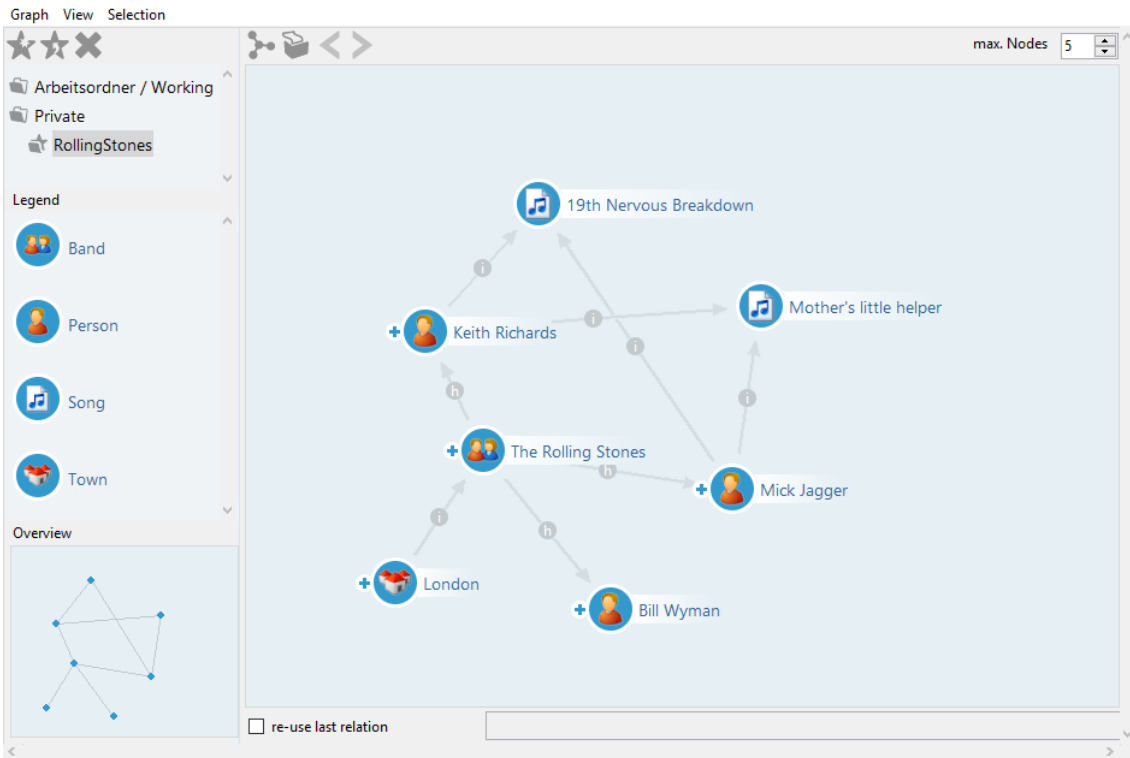
The menu *graph* contains more functions for the graph editor:

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.

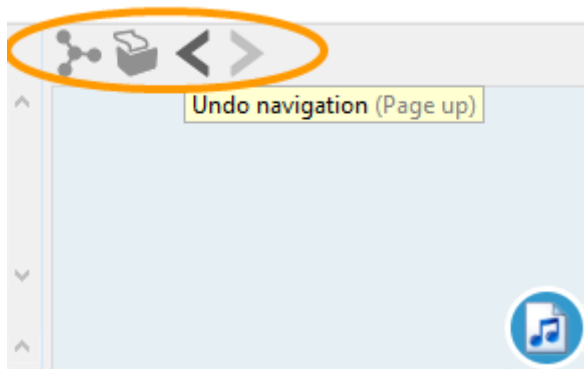



When a bookmark is created it may be given a name. All nodes contained in the bookmark are listed in the description of the bookmark.

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



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".



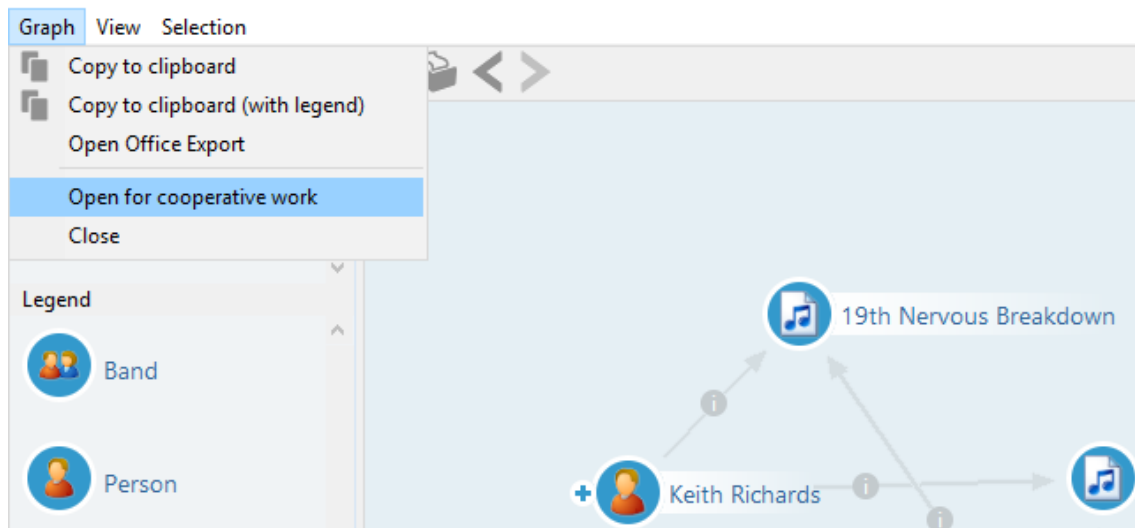
Layout: the layout function  enables you to position nodes automatically within the display area at the currently selected zoom level when many nodes are not allowed to be positioned manually. When more nodes are displayed they will also be automatically positioned in the graph via the layout function.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

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 programme, for example.



Print: opens the dialogue window for printing or for generating a pdf file from the displayed graph.

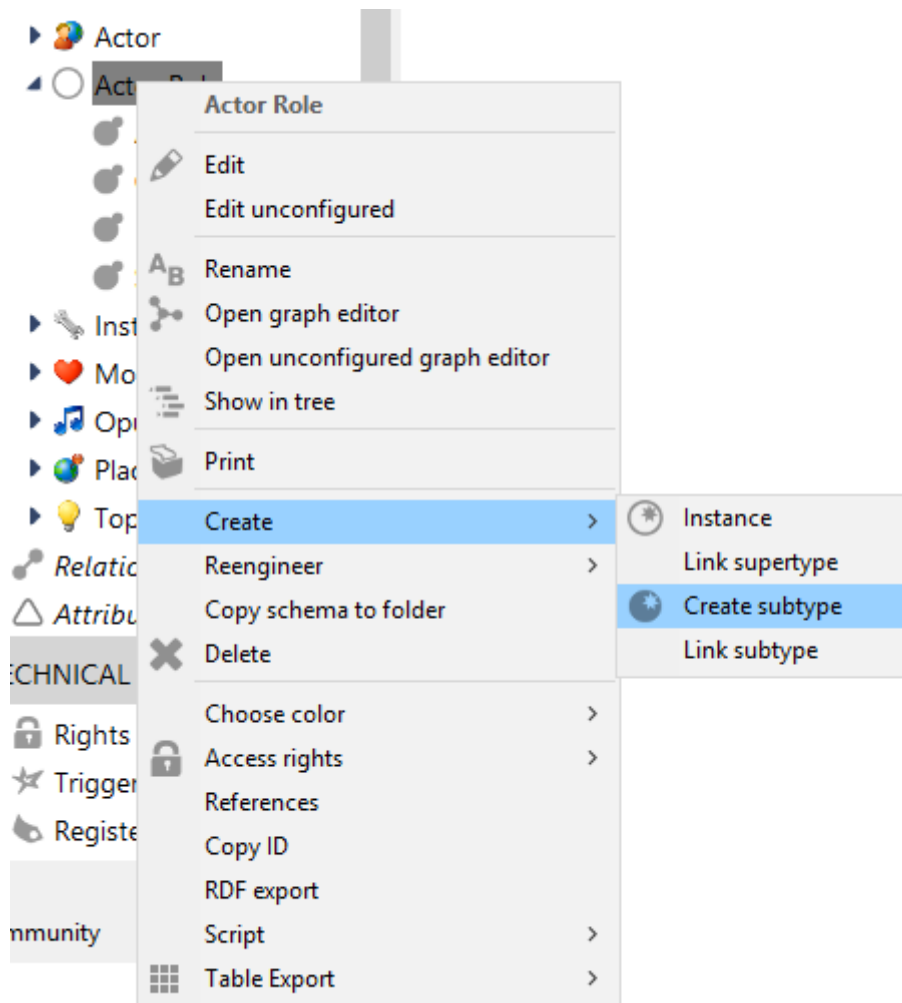
Cooperative work: this function enables other users to work on the graph mutually and simultaneously. All changes and selections of a user on the graph (layout, showing/hiding nodes, etc.) will then be shown to all other users synchronously.



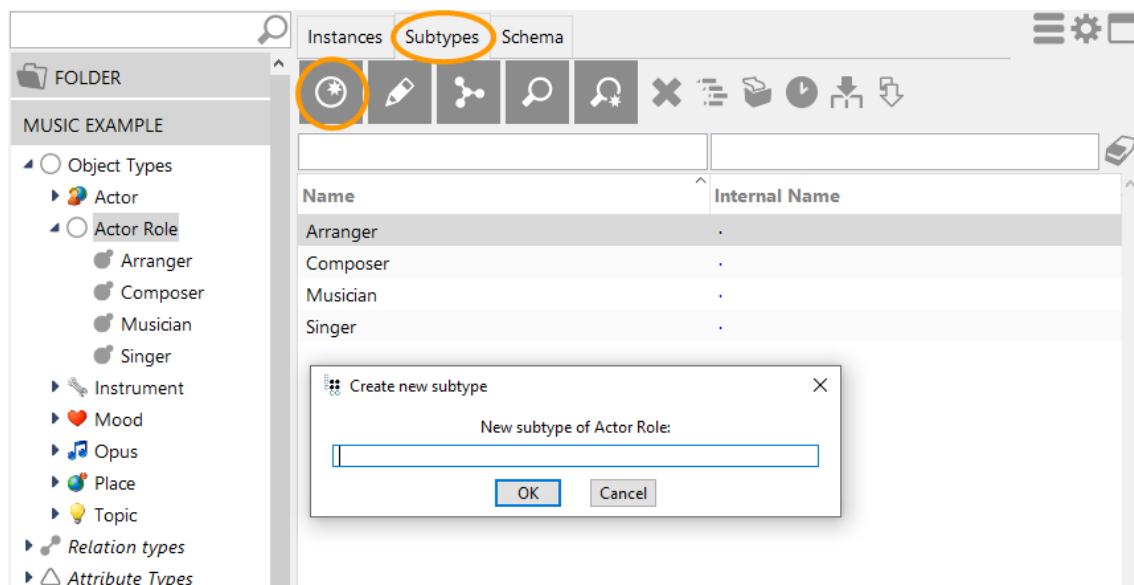
1.2 Definition of schema / model

1.2.1 Define types

The principle of the type hierarchy was already presented in Chapter 1.1.2. 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":

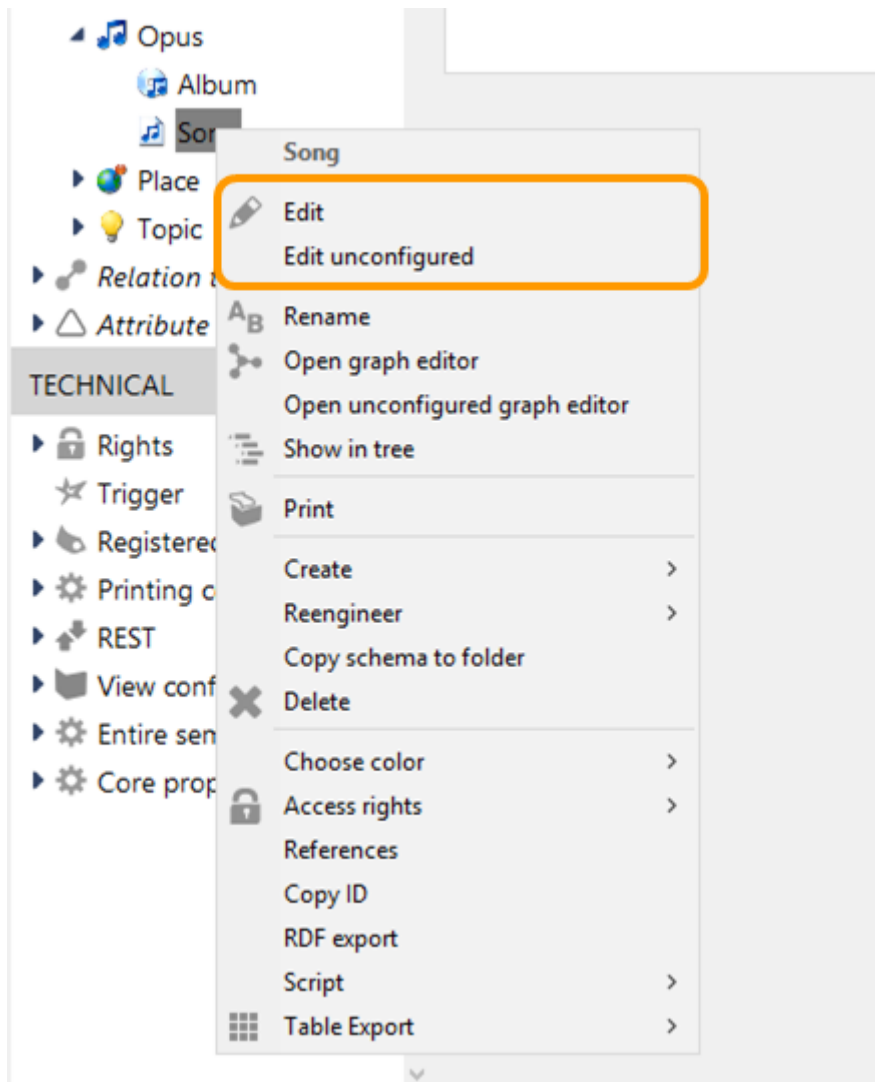




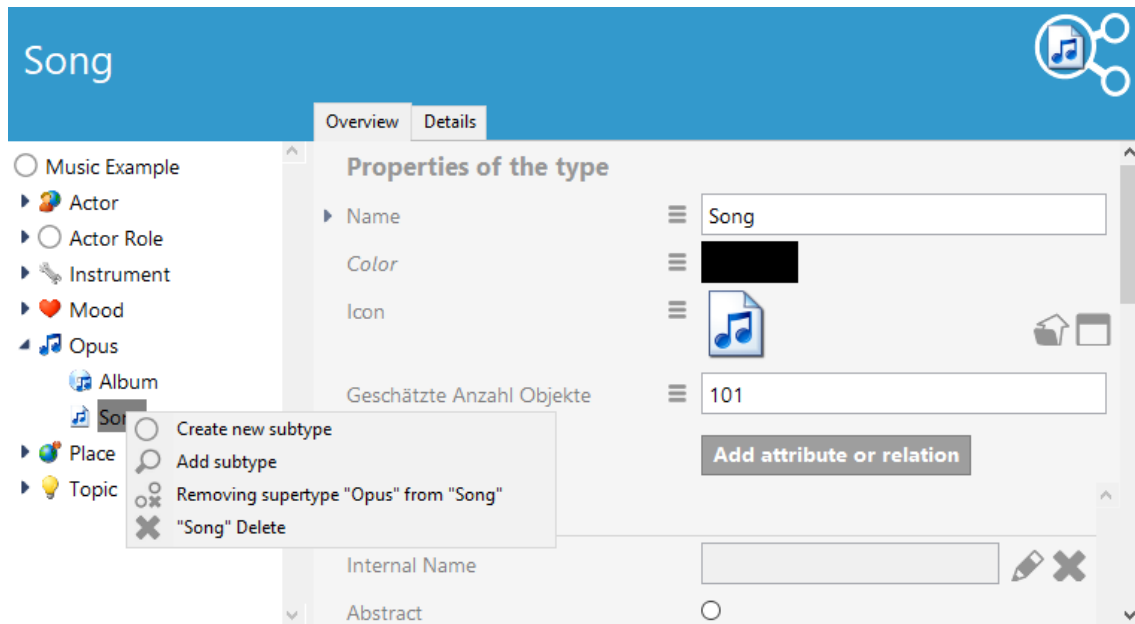
Changing the type hierarchy

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

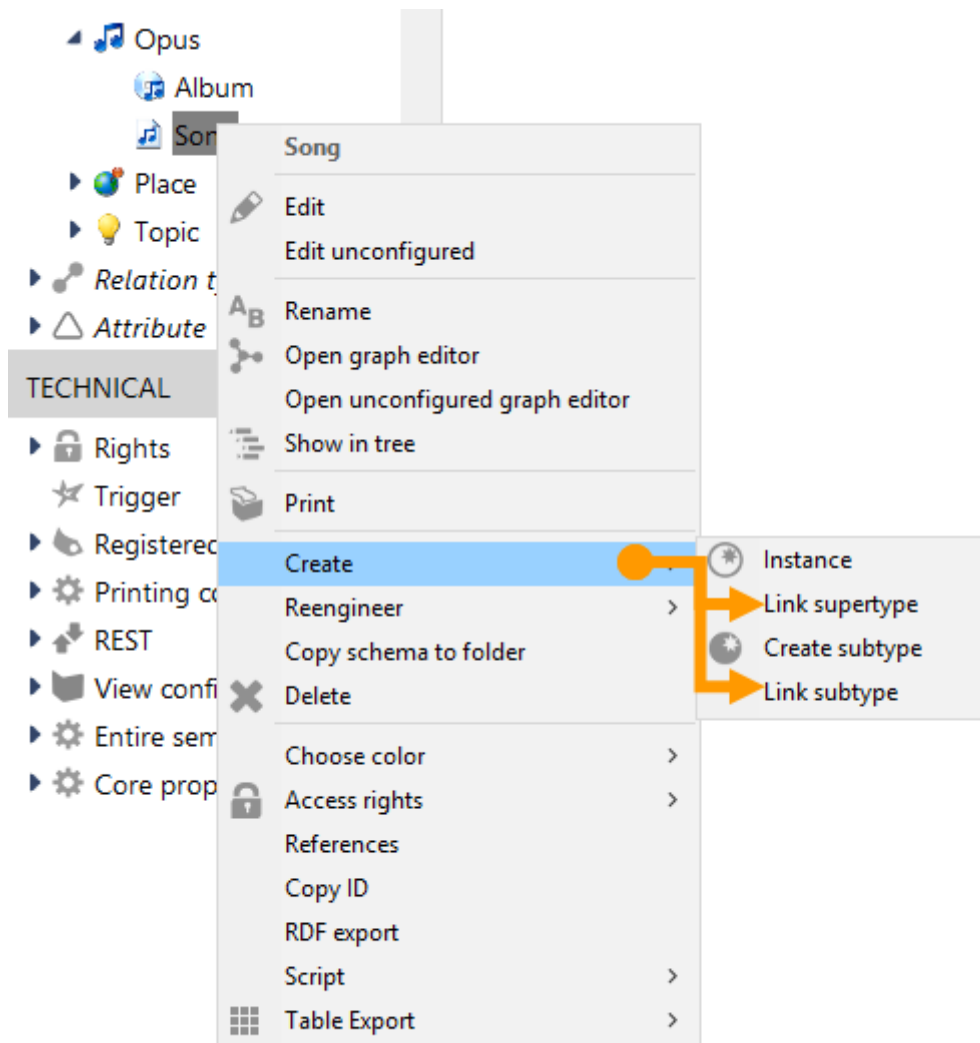
We also can change type assignment when opening the detail editor of the affected type by choosing the options "Edit" or "Edit unconfigured" in the context menu:



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

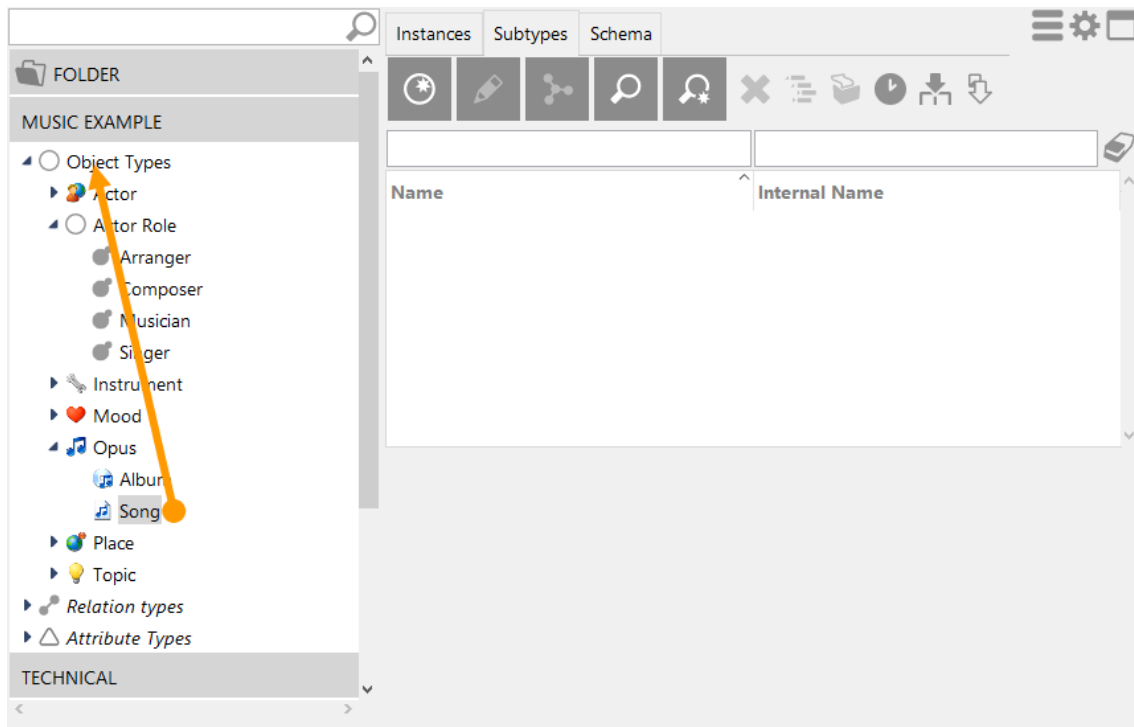


Using this option we can remove the currently selected object type from its position in the hierarchy of the object types. In the organizer, we can link types to other types in order to create multihierarchical schema:



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.

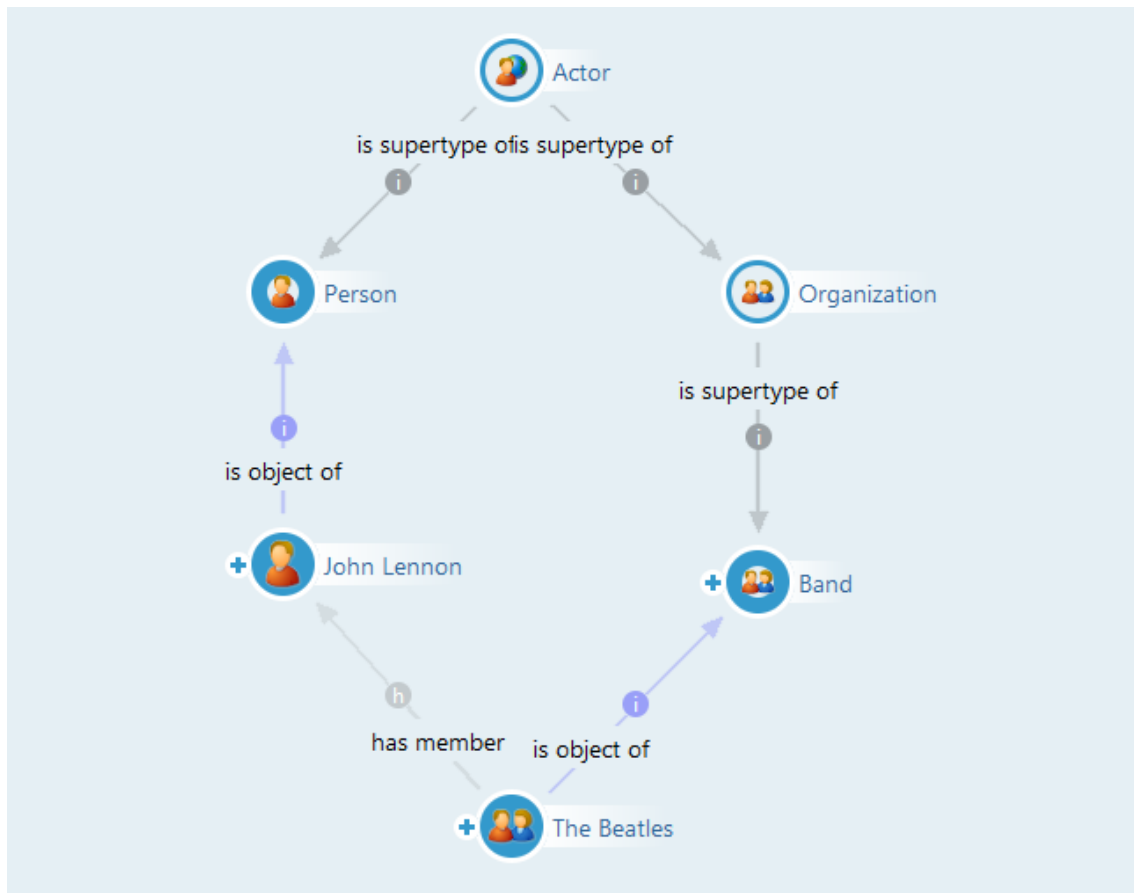
What still applies is: the hierarchy of the object type allows multiple assignments and inheritance.



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:



Band

- Music Example
- Actor
 - Organization
 - Band**
 - Person
 - Actor Role
 - Instrument
 - Mood
 - Opus
 - Place
 - Topic

OverviewDetails

Inherited Attributes

Define new attribute type

relations of objects

is author of	Instances of Opus
is band of	Instances of Musician
is performer of	Instances of Opus

Inherited Relations

Context element of	Instances of Static Tree Node, Instances of	> Top-level type
has genre	Instances of Music Genre	> Actor
has member	Instances of Person	> Organization
has place	Instances of Place	> Actor
is performer of	Instances of Album	> Actor

Define new relation type

Extensions

Add extension

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



The Beatles

Band

Attributes

▶ Name

The Beatles

Add attribute

Relations

is performer of

Abbey Road

has member

George Harrison

has member

John Lennon

has place

Liverpool

has member

Paul McCartney

has member

Ringo Starr

is band of

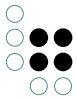
Ron Carter (Musician)

Add relation

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".





Type of relation	with own inverse relation		▼
	Relation	Inverse relation	
Name	is member of		has member
Supertype	User relation ...		User relation ...
Domain	Instances of Person ...		Instances of Band ...
Internal Name			
virtual	<input type="checkbox"/>		<input type="checkbox"/>
	Create		Cancel

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.



Choose attribute value type

Attribute

Boolean

Choice

Color value

Date

Date and time

File

Flexible time

Float

geo position

Group

Integer

Internet shortcut

Interval

Password

Reference to Mapping of a data source

Reference to Organizing folder

Reference to Query

Reference to Script

Reference to Semantic elements folder

String

Time

OK Cancel



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:

Attribute name	<input type="text" value="Stage Name"/>
Supertype	<input type="text" value="Attribute"/> ...
Defined for	<input type="text" value="Instances of Person"/> ...
Internal Name	<input type="text"/>
<input type="checkbox"/> May have multiple occurrences	

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 Create 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.

Type of relation	<input type="text" value="with own inverse relation"/>	
	Relation	Inverse relation
Name	<input type="text"/>	<input type="text"/>
Supertype	<input type="text" value="User relation"/> ...	<input type="text" value="User relation"/> ...
Domain	<input type="text"/> ...	<input type="text"/> ...
Internal Name	<input type="text"/>	<input type="text"/>
virtual	<input type="checkbox"/>	<input type="checkbox"/>

Editor for creating a new relation type (see also Chapter 2.1 Defining types)

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 accomodate 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 "writed 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.

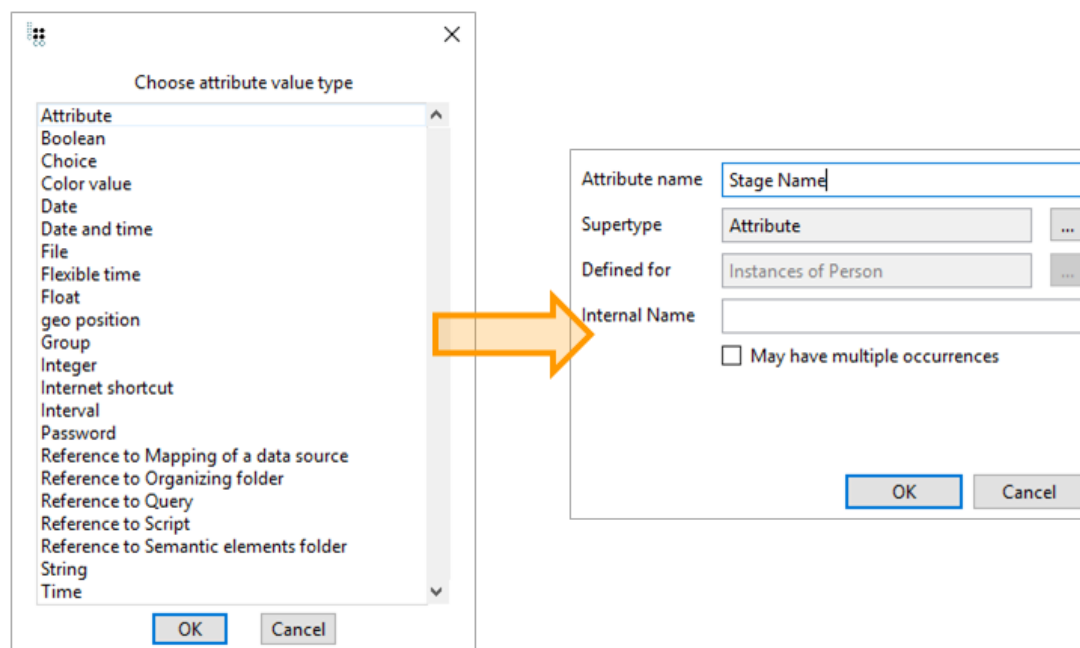
Domain: Here we define by which object types the relation has to be created: one object type forms the source of the relation and another object type the target. The tareget object type, in turn, forms the definition area of the inverse relation. To simplify matters, when creating you may only enter 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 refrence.

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 Create a new attribute type

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



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 (hollowbody, fretless, etc.)
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
Group	without attribute rating, 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	link on a 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	per attribute entity and password a clearly hashed value (Chaum-van Heijst-Pfitzmann) which is only used to validate the password	
Reference to [...]	reference to parts of the Knowledge Graph configuration: search, diagram of a data source, scripts and files - is used for example in the REST configuration	
String (character string)	random sequence of alphanumeric characters	review text to a recording medium
Time	time hh:mm:ss	duration 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.

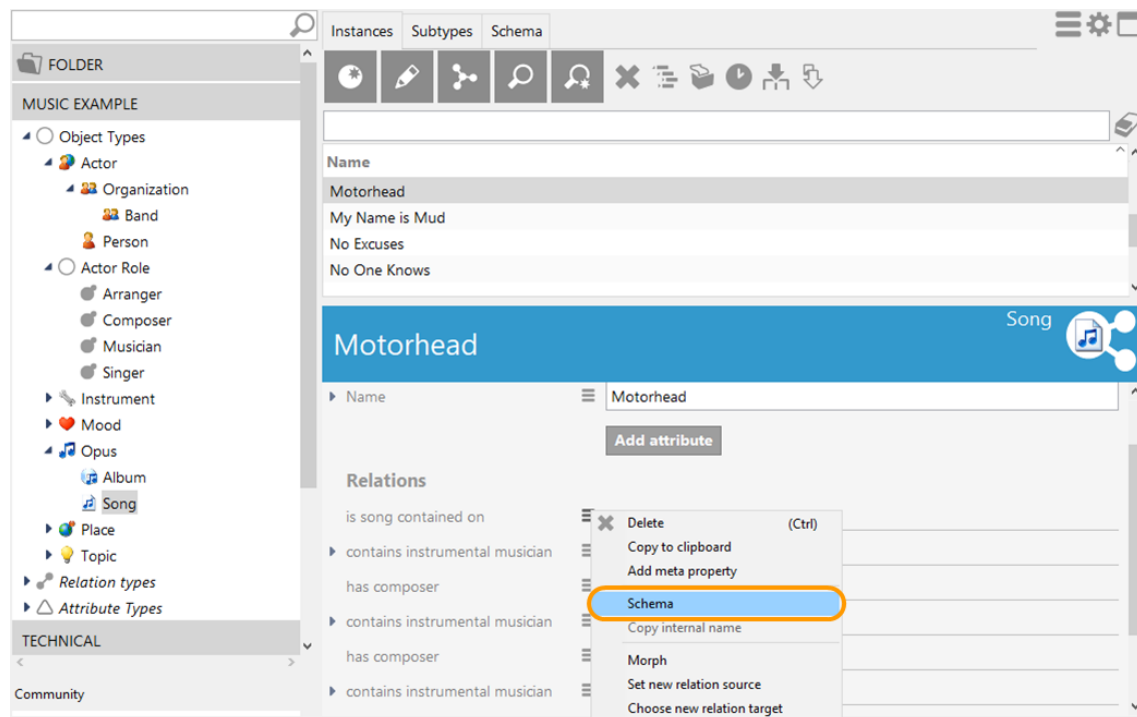



May have multiple occurrences: attributes may occur once or more than once, depending on the attribute type: a person only has one date of birth but may, for example, have several academic titles at the same time (e.g. doctor, professor and honorary consul).

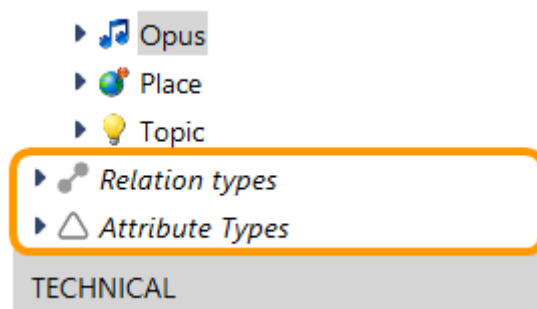
1.2.2.3 Edit details

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 an 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):



has place

Overview Details

Relation

- Inferred relation
- System relation
- User relation
 - authentication
 - contains guest appearance
 - contains song
 - correlates with
 - correlates with (inverse)
 - has author
 - has cover version
 - has geographical part
 - has guest appearance
 - has member
 - has partner
 - has performer
 - has performer
 - has place
 - has remixed version
 - Input media type
 - is authentication of
 - is author of
 - is band of
 - is cover version of
 - is geographical part
 - is input content type
 - is member of
 - is musician in
 - is output MediaType
 - is performer of
 - is performer of

Icon

average number (calculated)

estimated number of instances

is property of

has place

Property

Add attribute or relation

Definition

Internal Name

Defined for

Instances of Actor

Target

Instances of Place

Inverse relation type

is place of

Abstract

May have multiple occurrences

Mix-In

Single-sided relation

Main direction

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.

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: (Note: can only be set on object types, not relation or attribute types)



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.

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.

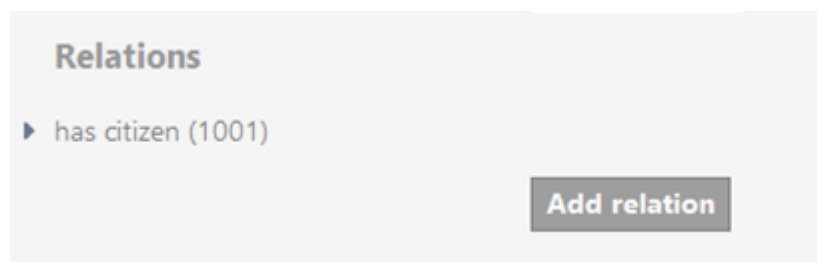
minOccurs guideline: This reference value relates to the user interface in Knowledge Builder and as of Version 5.3 it also affects the user interface in the web front-end and specifies the minimum number of times a property is supposed to occur on an object. If the number falls below the specified number, 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: As of Version 5.3, this reference value relates to the user interface in Knowledge Builder and the user interface in the web front-end. It 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

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.

Example: 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.

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.

New relation type

Type of relation: with own inverse relation

	Relation	Inverse relation
Name	is citizen of	has citizen
Supertype	User relation	User relation
Domain	Instances of Person	Instances of City
Internal Name		
virtual	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Create Cancel

Single-sided relation = „visible“/real relation half

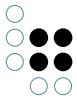
„Invisible“/virtual relation half

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.



is citizen of

Overview Details

Properties of the type

- Name: is citizen of
- Color: [Black]
- Icon: [None]
- Average quantity (computed): [None]
- Estimated number of objects: [None]

Definition

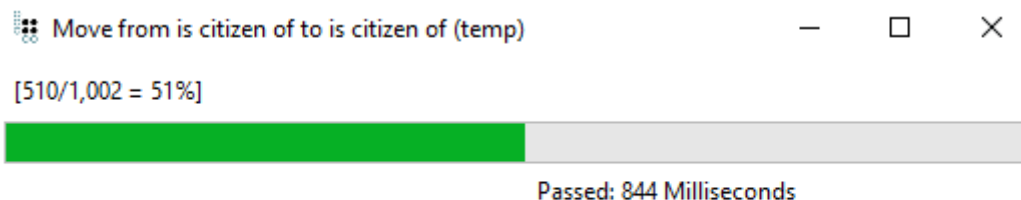
- Internal Name: citizenOf
- Defined for: Instances of Person

Context menu options:

- Edit
- Edit unconfigured
- Copy internal name
- Rename
- Open graph editor
- Open unconfigured graph editor
- Show in tree
- Print
- Create
- Reengineer
- Delete
- Access rights
- References
- Copy ID
- Script
- RDF export

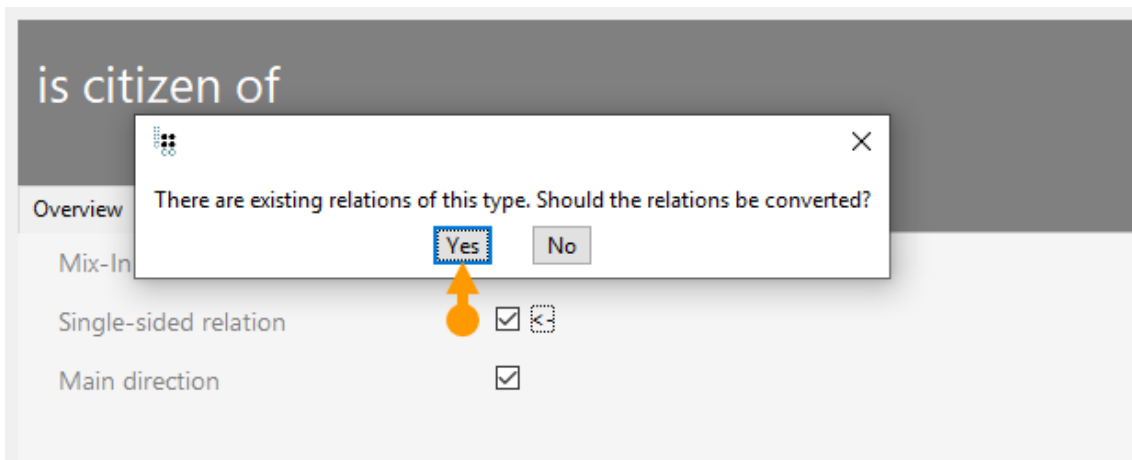
Additional options in context menu:

- Delete all elements
- Morph
- Convert to One-Way-Relation
- Add attribute or relation



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

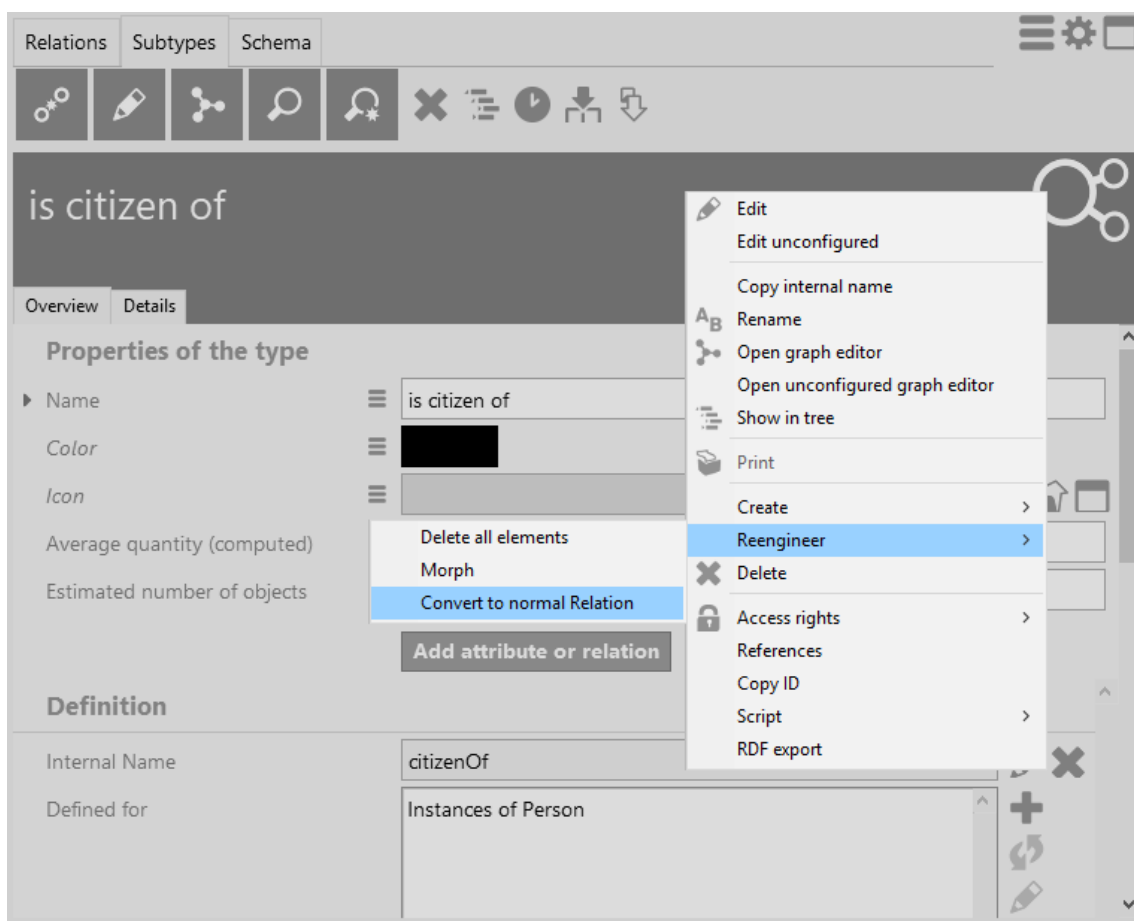
Hint: Until i-views 5.3, the checkbox of the Boolean attribute "Single-sided relation" only served for indication purpose. Since i-views 5.4, a redefinition only can be executed by clicking on the checkbox **or** via the context menu in the detail editor.



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

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.

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:



Delete the following objects?

▼ John Lennon

Name: John Lennon

▼ John Lennon is composer of Come Together

Come Together has composer John Lennon

▼ John Lennon plays instrument Guitar

Guitar is played by John Lennon

▼ John Lennon is member of The Beatles

The Beatles has member John Lennon

▼ John Lennon is instrumental musician on Come Together

▼ (Come Together contains instrumental musician John Lennon) song is played by

Electric Piano is played by musician on song (Come Together contains instrum

▼ (Come Together contains instrumental musician John Lennon) song is played by

Guitar is played by musician on song (Come Together contains instrumental n

Come Together contains instrumental musician John Lennon

▼ John Lennon is vocalist on Come Together

<

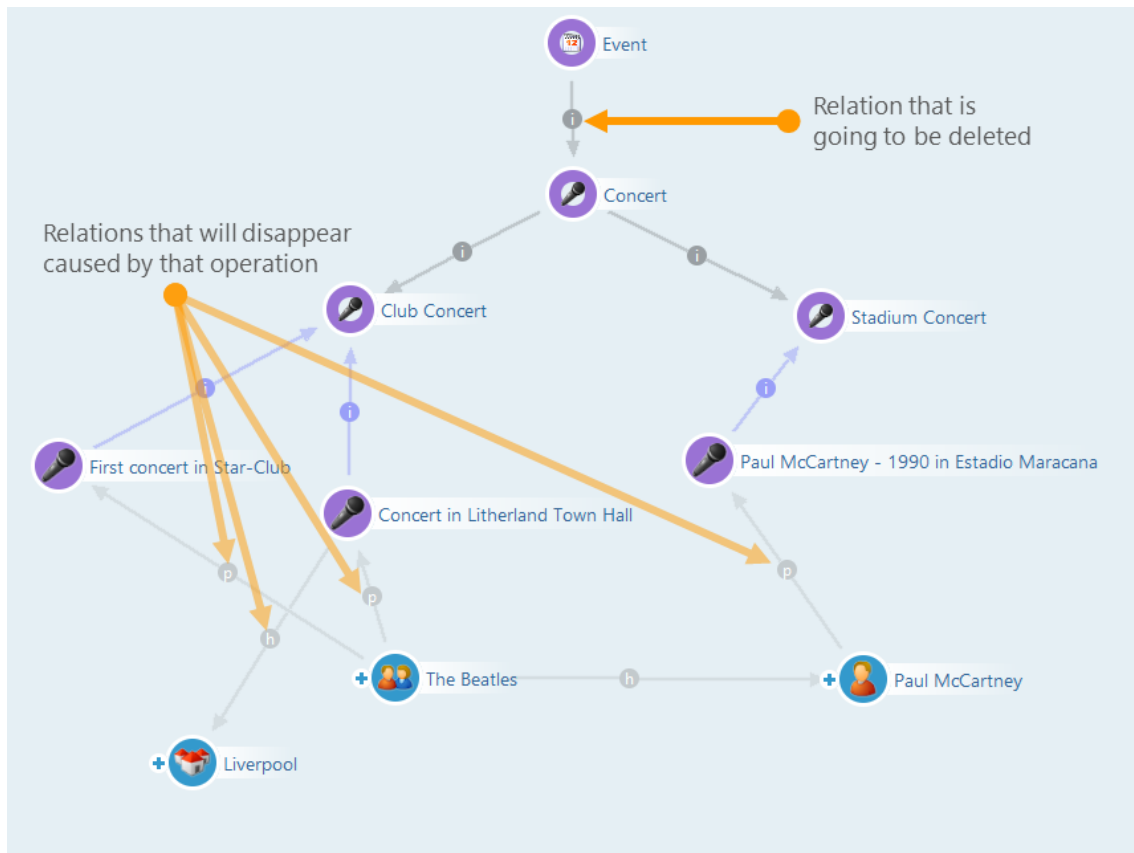
>

OK

Cancel

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.

Special functions

Changing type: objects already in the Knowledge Graph may be moved to objects of another



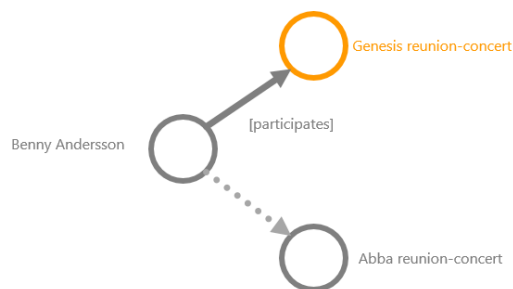
type. For example, if the object type "event" differentiates to "sports event" and "concert". If there are already objects of the type sports event or concert in the Knowledge Graph, they may be selected from the list in the main window and quite simply moved to a new, more suitable object type using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

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

Select type: using this operation we can assign a property to an object.



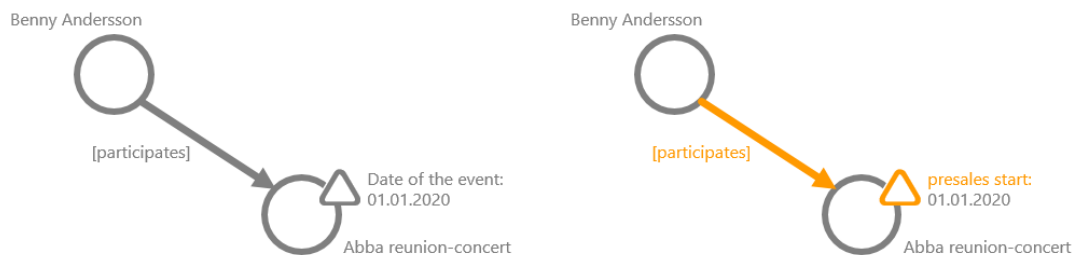
Reselect relation target: in relations this does not only apply to the source, but also the relation target.



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.

Converting the relation: source and target of the relation will remain the same, only the relation type will be converted.

Converting the attribute: source/object will remain the same but it will be assigned to another attribute type:



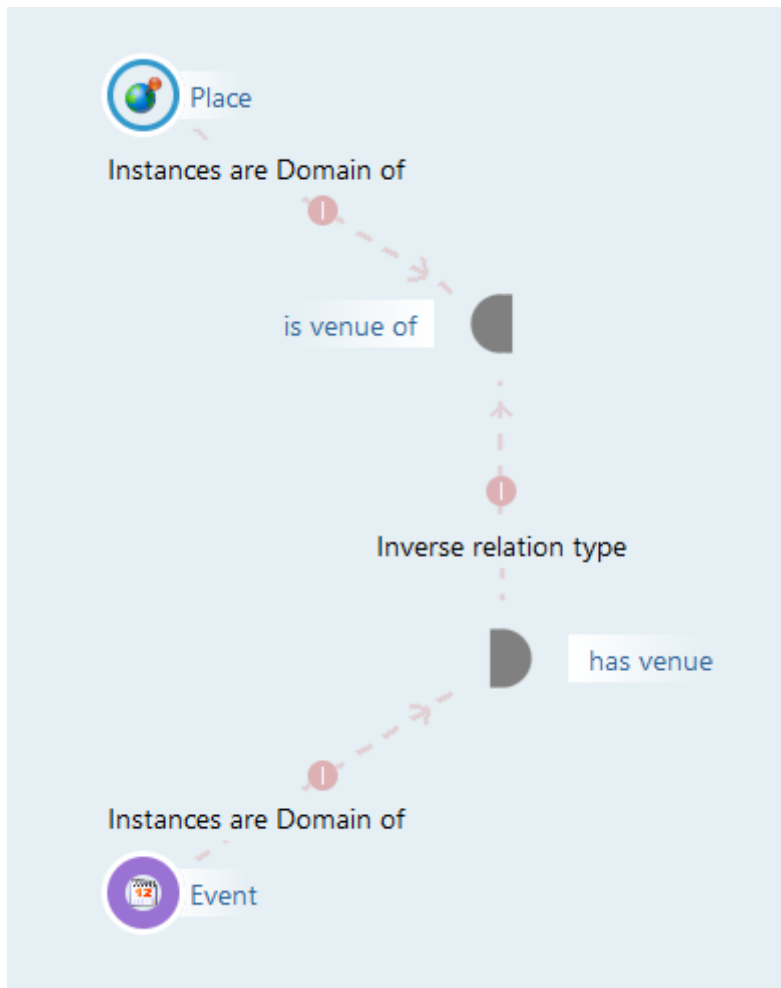
When converting the individual relations we are usually quicker when we delete these and replace them with another one. However, it may happen that meta properties are attached to the properties which we do not want to lose. On the other hand, the converting operations are also available for all properties 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.

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.

1.2.4 Representation of schema in the graph editor

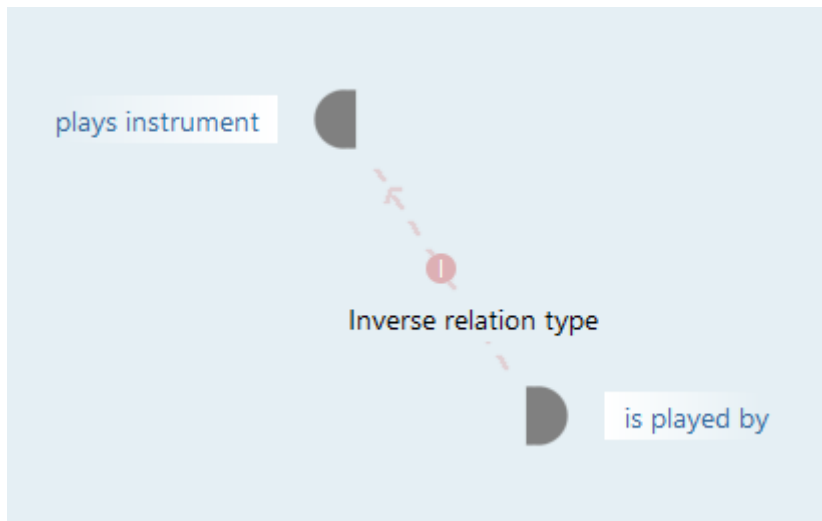
Until now we have mainly been dealing with linking of specific objects within the graph editor. Presenting such specific examples, discussing them with others and, where necessary, editing them is also the main function of the graph editor. We can, however, also present the model of the Knowledge Graph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

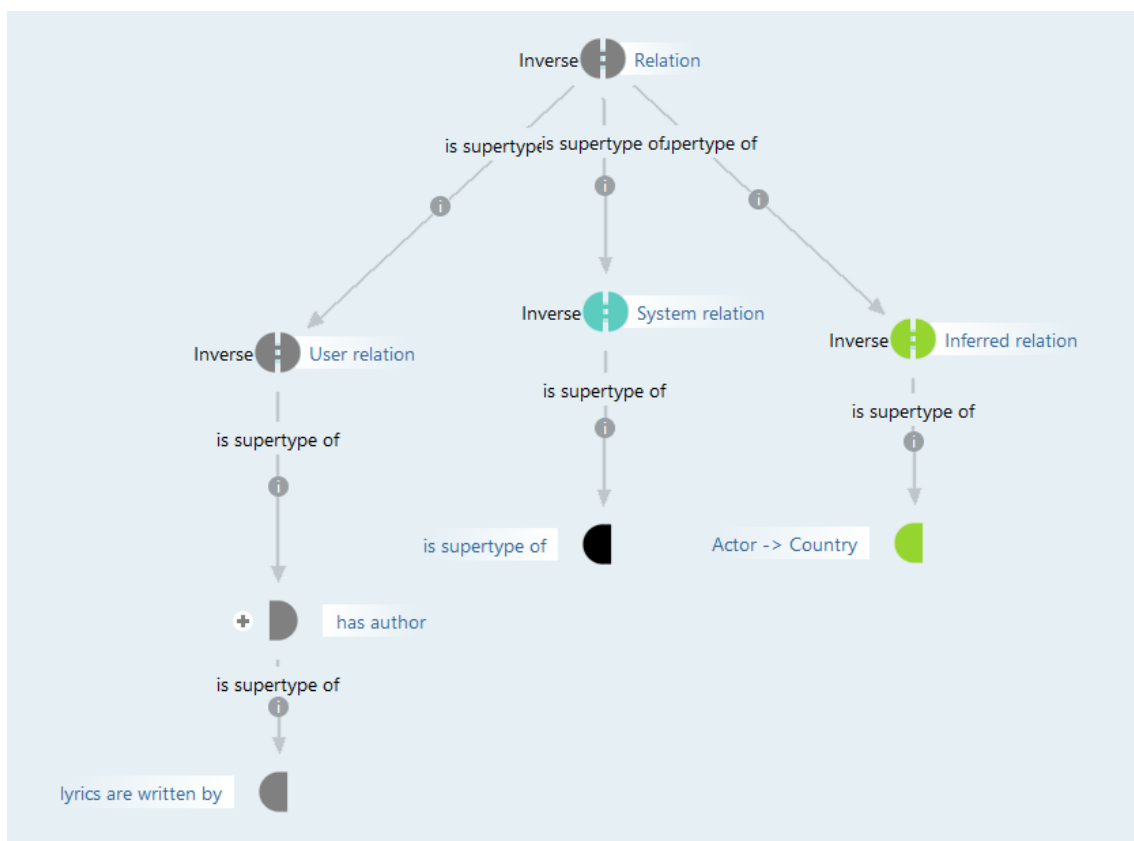


Relation types in the graph editor

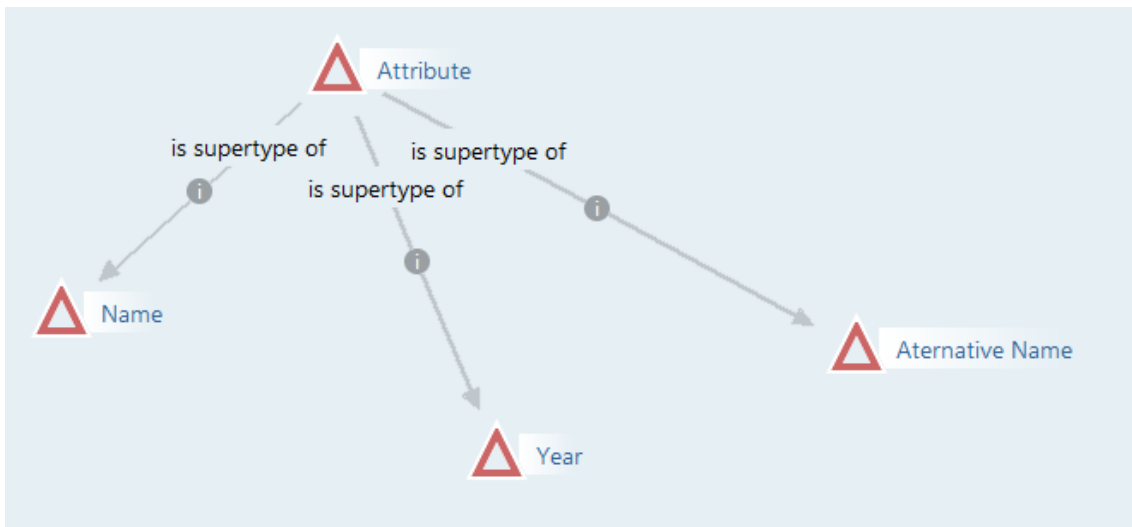
If until now we have been referring to relations in the graph editor, this concerned relation objects between specific objects of the Knowledge Graph. Moreover, the general types of relations (hence the diagrams of the relations) may also be presented in the graph editor. A relation is depicted in the 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.



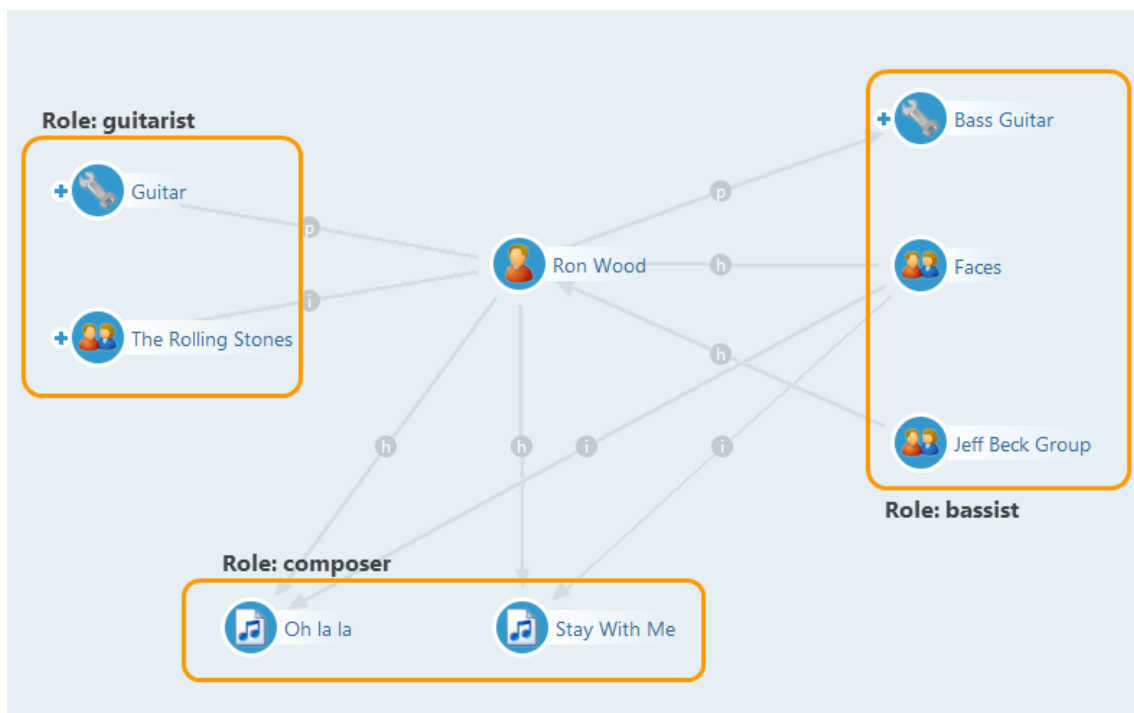
Analogue to the type of object hierarchy the hierarchy of the relations and attributes within the graph editor may be changed by deleting and dragging the supertype relation.

1.2.5 Metamodeling and advanced constructs

1.2.5.1 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":

Guitarist

Overview Details

Properties of the type

Name: Guitarist

Color: [Black]

Icon: [Upload]

Add attribute or relation

Definition

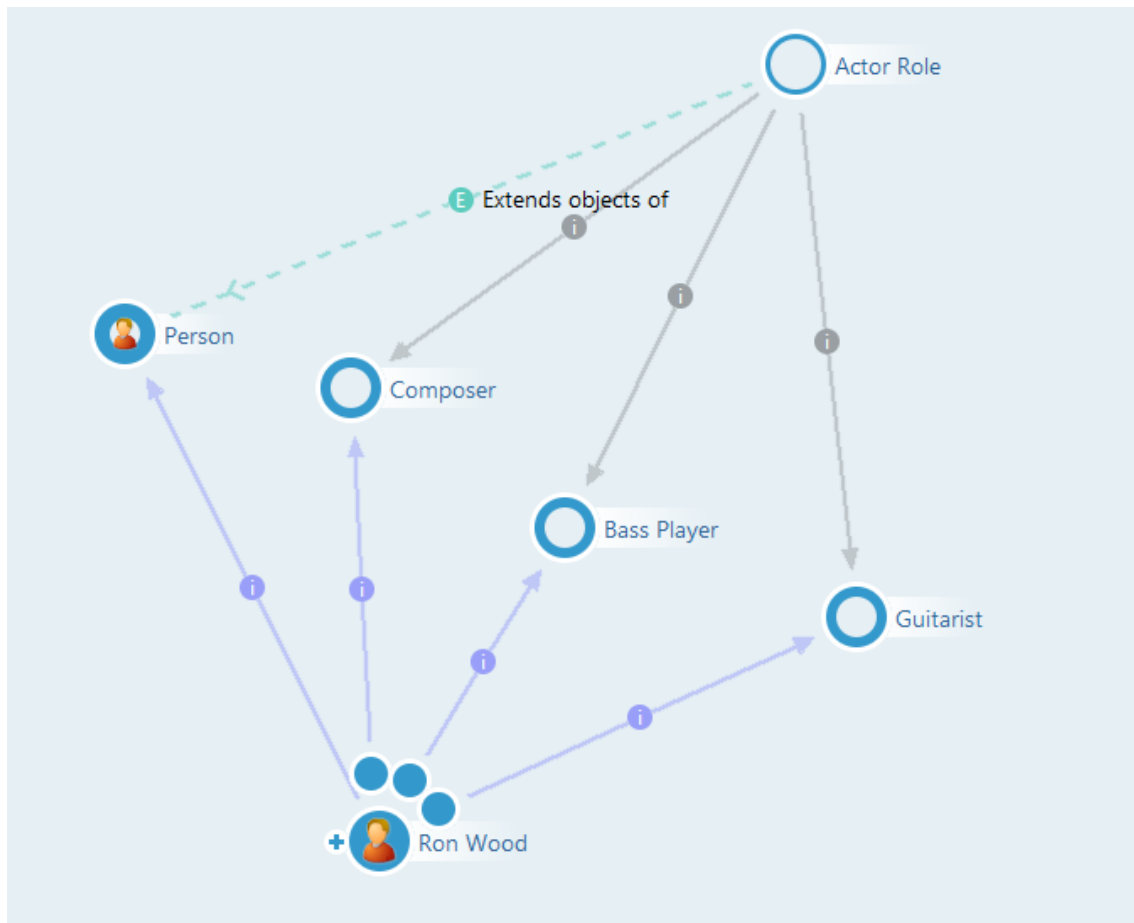
Internal Name: [Input] [Edit] [Delete]

Abstract: ☐

Type is not abstract: ☐

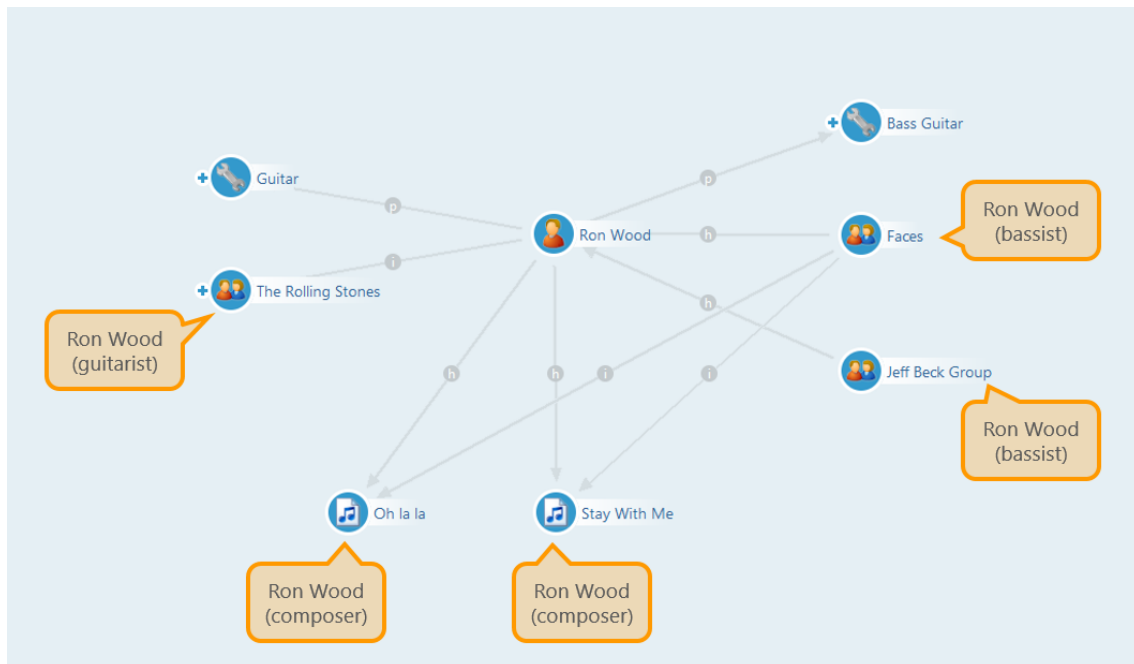
Type can extend objects: ☒

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



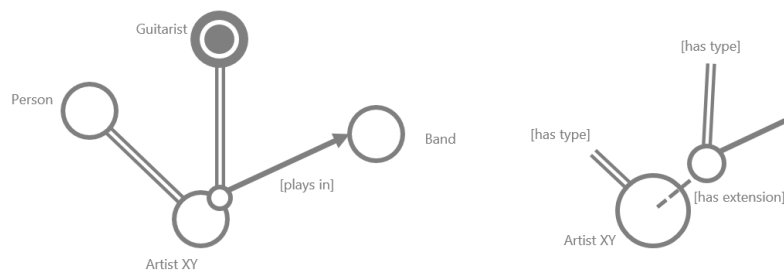
As a result of this enhancement we have achieved several things simultaneously:

- 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".



The screenshot displays a graph and a structured query interface. The graph shows two nodes: 'The Rolling Stones' (a band) and 'Ron Wood' (a person). They are connected by a relation labeled 'plays in band'. The structured query interface shows a query with 'Person' as the source, 'Guitarist' as the target, and 'plays in band' as the relation. The 'Query Result' table shows 'Ron Wood' as a 'Guitarist' and 'The Rolling Stones' as a 'Band'.

Name	[3] Guitarist	[5] Band
Ron Wood	Ron Wood (Guitarist)	The Rolling Stones

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.

The screenshot displays a graph and a structured query interface. The graph shows two nodes: 'The Rolling Stones' (a band) and 'Ron Wood' (a person). They are connected by a relation labeled 'has member'. The structured query interface shows a query with 'Person' as the source, 'Guitarist' as the target, and 'plays in band' as the relation. The 'Query Result' table shows 'Ron Wood' as a 'Guitarist' and 'The Rolling Stones' as a 'Band'.

Name	[3] Guitarist	[5] Band
Ron Wood	Ron Wood (Guitarist)	The Rolling Stones



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.

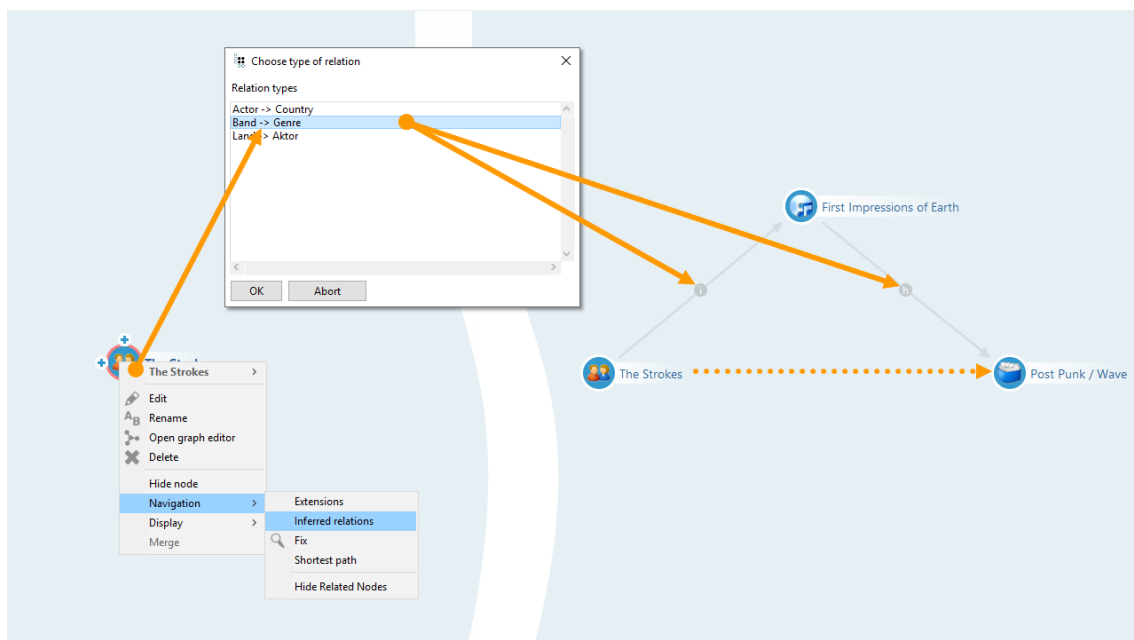
If a mix-in enhancement is deleted it acts like a "normal" enhancement: there has to be at least one enhancement available which entails the mix-in property. When the last of these enhancements is deleted the relation or the attribute in the core object is also deleted.

1.2.5.2 Inferred relations

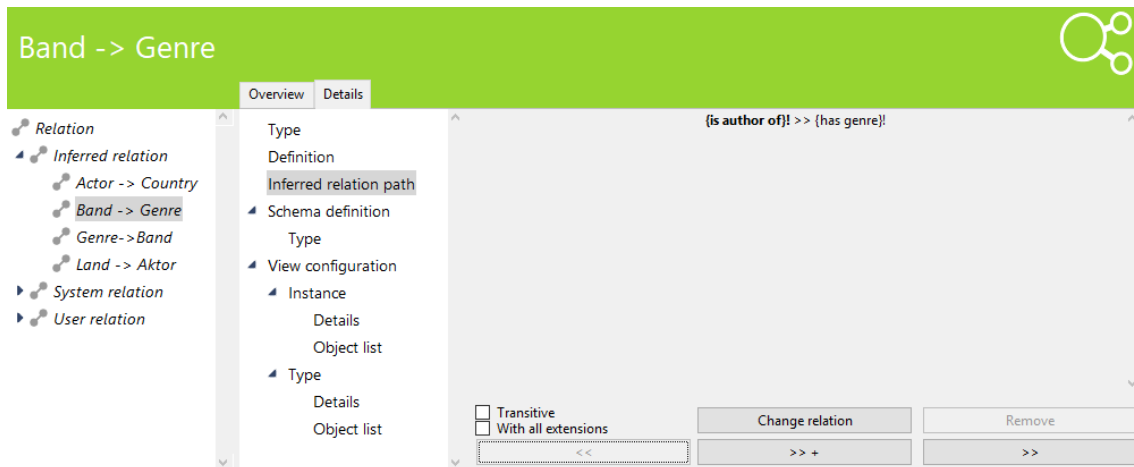
A special form of the relation is the shortcut relation. Hidden behind this is the possibility to shorten several relations already available by means of schematically predefined substitute relations.

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.

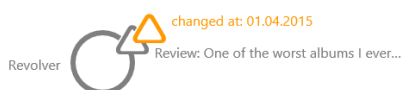
In the queries the shortcut relation can be used like any other relation as well.

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.

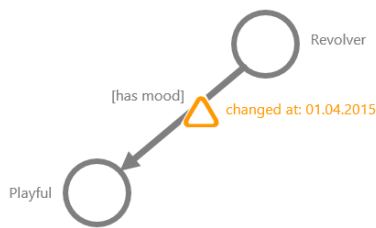
1.2.5.3 Meta properties

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.

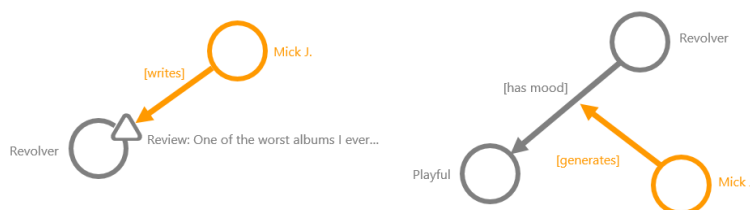
Attributes to attributes: for example, discussions and reviews are listed in the music Knowledge Graph as text attributes for music albums. If it is to be noted when discussions and reviews were added or when they were last changed we can define a date attribute which is assigned to the discussion and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

1.2.5.4 Multilingualism

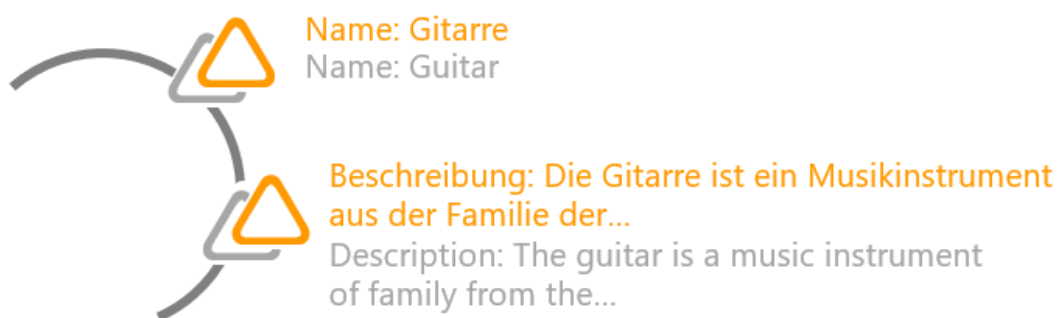
The attributes "character string", "data file attribute" and "selection" may be created multilingually. In the case of the character string attribute and data files, several character strings may then be entered for an attribute:

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.

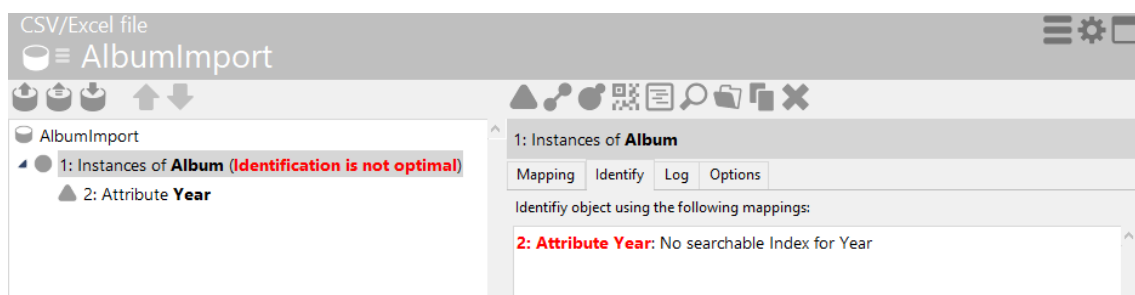


1.2.6 Indexing

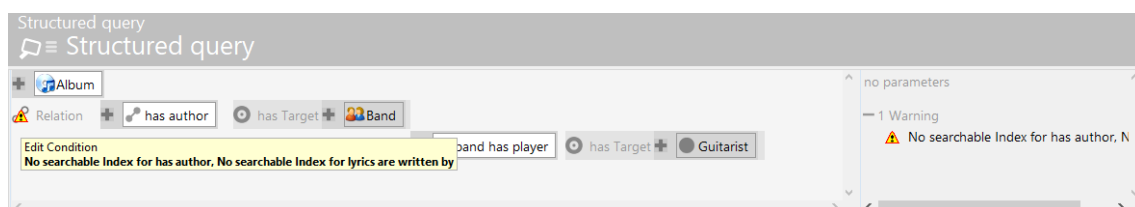
Indexing forms part of the internal data management of databases. Used correctly, the setting of indexes can improve performance significantly.

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 can improve performance in particular when it comes to writing data (= importing).

Indexing is required for:

- Transaction:
Read transactions: Search/structured query; view configuration
Write transactions: Imports (import mapping)
- Checking rights

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

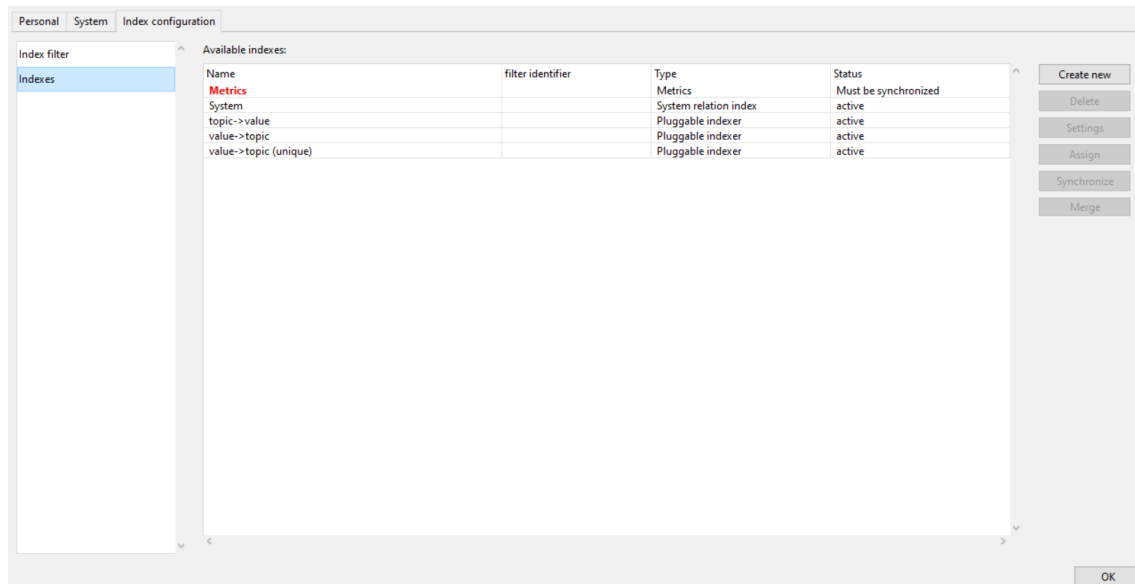
The indexes are **defined** in the Knowledge Builder settings. The **assignment** of the indexes can take place either in the settings of the KB or in the Detail editor of a type (Details > Indexing > Assign index).



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 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 (write rights check for imports)
value -> topic (word)[string splitting]	CDP-specific: This is only used in <i>i-views content</i>
value -> topic for subject keys (word)[string splitting]	CDP-specific: This is only used in <i>i-views content</i>
Full text index for terms [string splitting]	CDP-specific: This is only used in <i>i-views content</i>

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

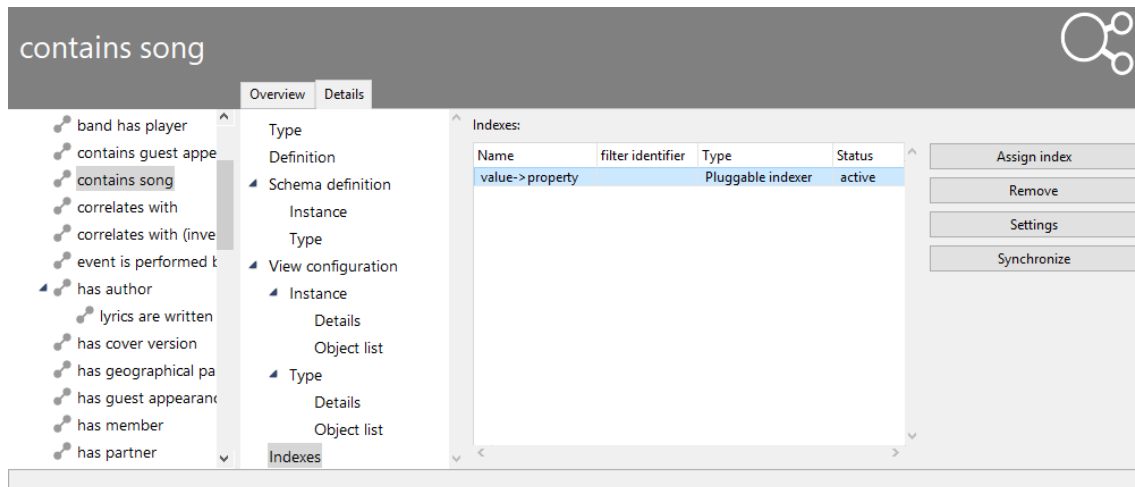
Indexes can be divided up using different aspects. First of all, a distinction can be made between forward and reverse indexes. In the case of the reverse indexes, it may make sense to refer to the property from target/value to resolve the metaconditions on the property. Ultimately, an index can optionally perform a segmentation by each type of source object in order to 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
Lucene full text index (JNI)	Full text query
Pluggable indexer	Combined use of distributor and index modules for adapted indexing; specific configuration by means of index filters is possible

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.

Addable index modules

Distributor by domain

Distributor by property type

Distributor by property value

Distributor by semantic element

Index Redundant Storage of Relation Properties

Add Index module

Assigned index modules

-???

Remove last index module

No filter

Select filter

filter identifier

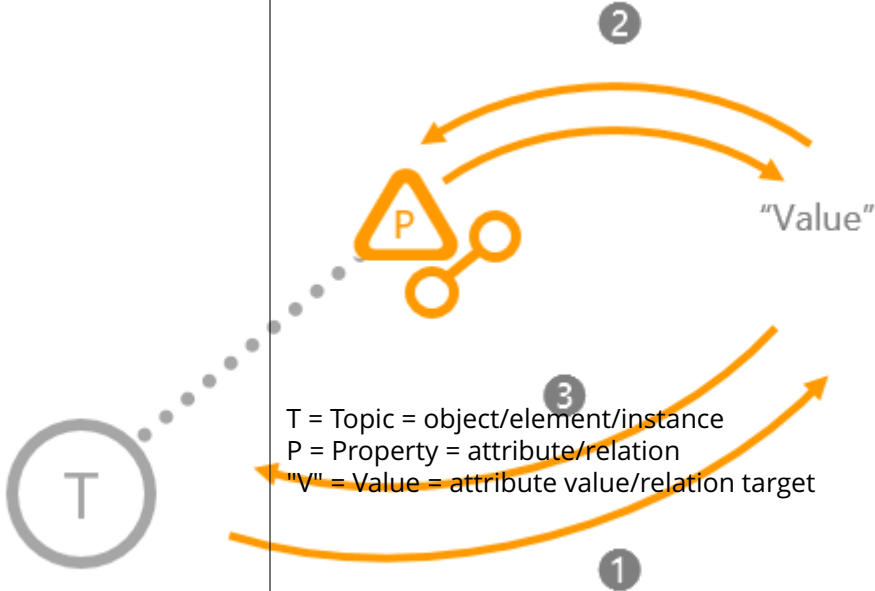
Indexer Name

Abort

OK

Pluggable indexer



	 <p>T = Topic = object/element/instance P = Property = attribute/relation 'V' = Value = attribute value/relation target</p>
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
Distributor for each property type (index can be selected afterwards:)	Distinction between attribute and relation
Index property on value/target	<p>Attribute -> Attribute value, Relation -> Target object/target type</p> <p>To find relation targets in structured queries with a restriction on the meta property</p>
Index object on value/target = topic -> value = topic -> value (domain segmented)	<p>Object -> Attribute, Object -> Target object of relation</p> <p>To list attribute values/relation targets in object lists</p>
Index value/target on property = value -> property	<p>Attribute value -> attribute Meta-relation target -> Attribute Relation target -> relation Meta-attribute (value) -> Relation</p> <p>For single-sided relations, results in a speed-up for weighted inverse one-way relations</p>



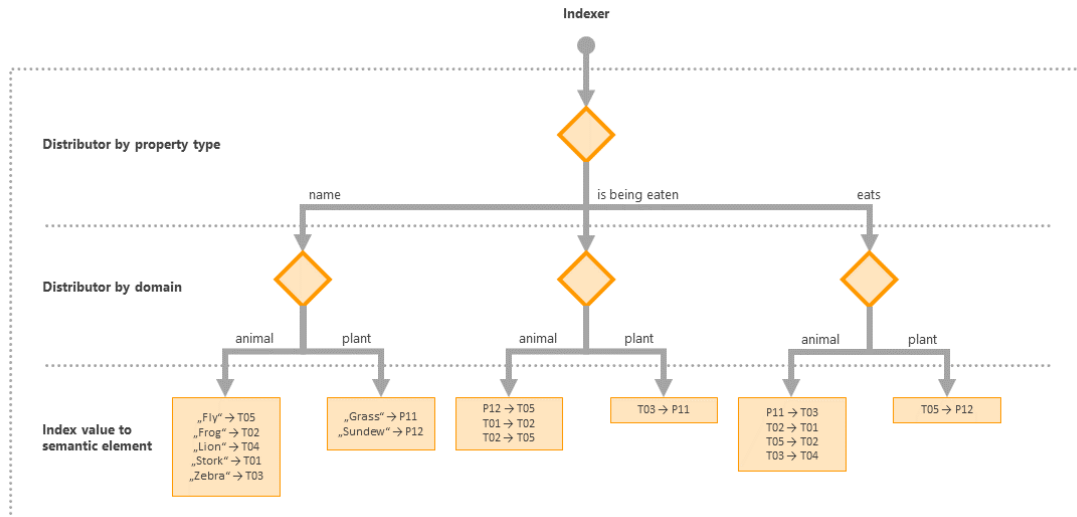
Index value/target on property (uniqueness check)	Attribute value -> Attribute To search for meta properties
Index value to semantic element = value -> topic	Attribute value -> Attribute Relation target -> Relation To support structured queries on objects with specified values/targets on attributes/relations
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"
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.)

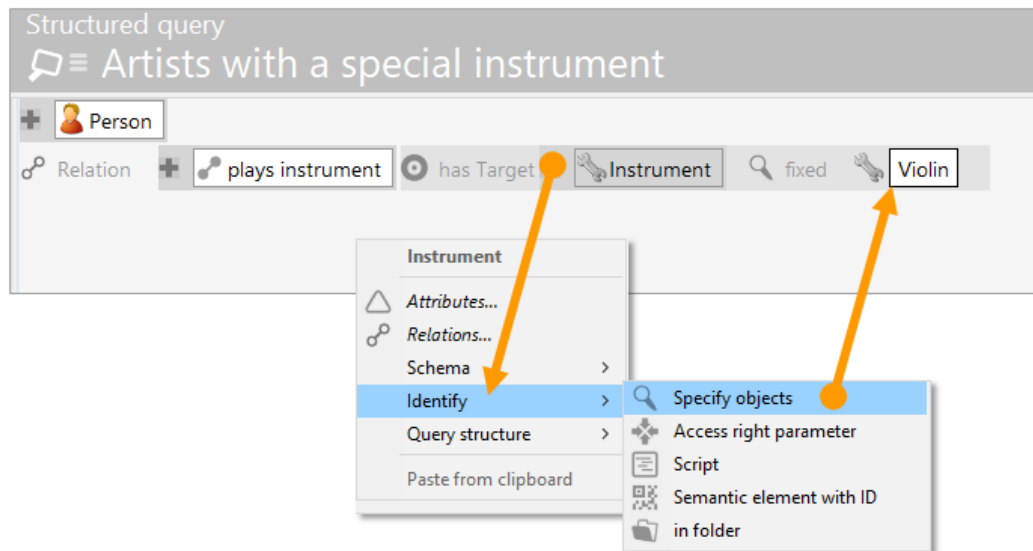
1.3 Searches/Queries

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.

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

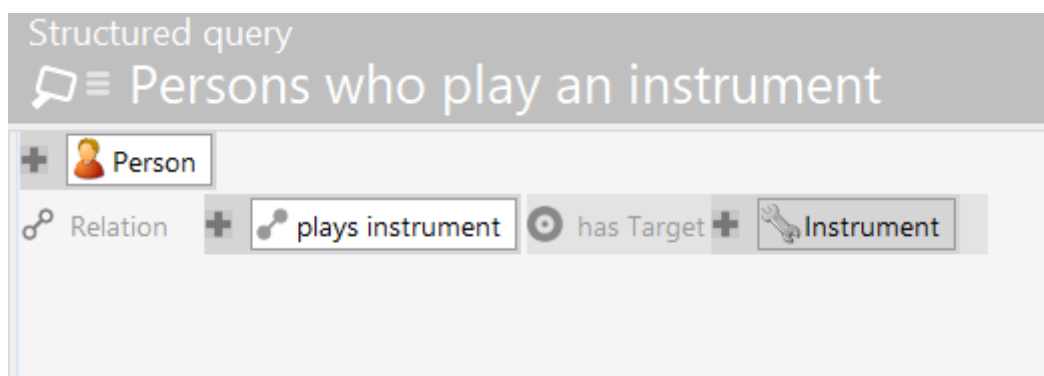
1.3.1 Structured queries

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 filtered.

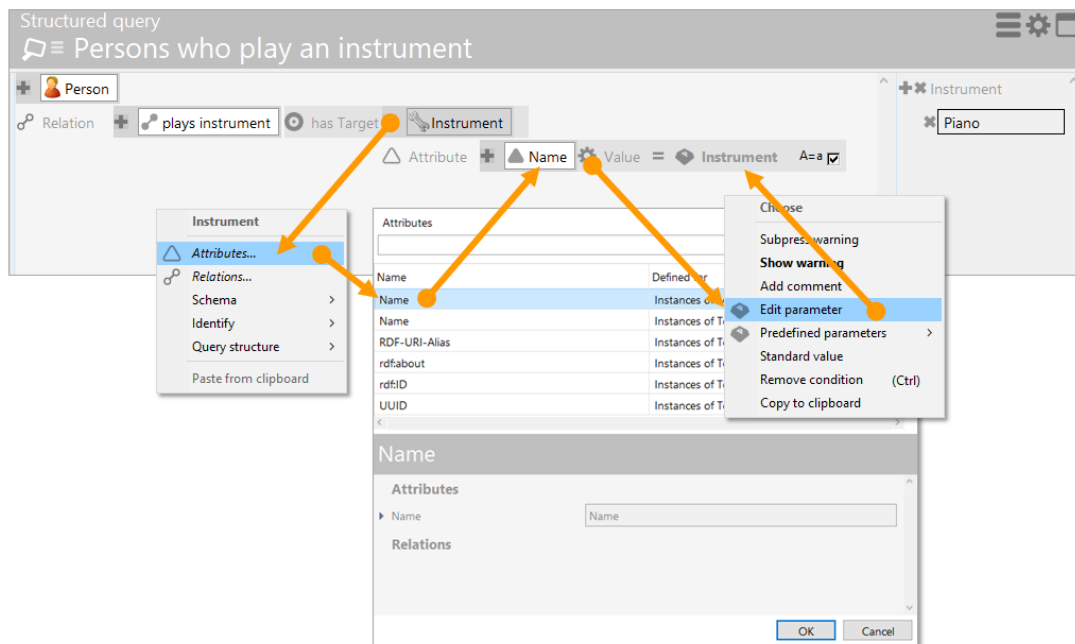


At first there is the type condition: objects of the type person are searched for. The second condition: the persons have to master an instrument. Third condition: this instrument has to be the violin.

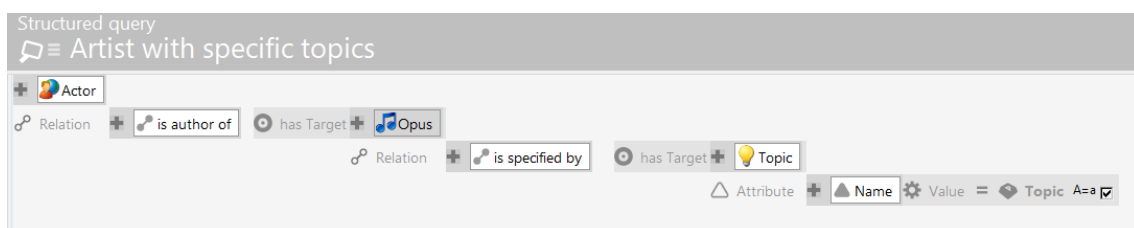
In the structured query the relation "plays instrument", the type of the target of the relation and the value of the target "violin" form three different conditions and thus also three search modes. The third condition that the instrument has to be a violin may also optionally be omitted. In the hit list you would then find all persons who play any random instrument.



Often conditions (in this case the instrument) should not be determined previously but be approved completely. Depending on the situation, an instrument may be given as a parameter in the application:



The conditions may thereby be randomly complex and the Knowledge Graph traversed as far as possible:

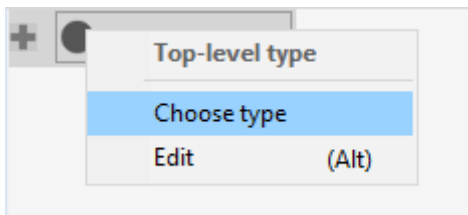
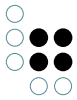



Slightly more complex example: persons or bands who deal with a certain issue in their songs (to be more exact in at least one). In this case you do not search for the name but the ID of the issue as the parameter - typical for searches, for example, which are queried via a REST service from the application [Figure - "ID" instead of "name"] or by a script.

The type hierarchies are automatically included in the structured queries: The type condition "Opus" in the search box above includes its subtypes albums and songs. Even the relation hierarchy is included: 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.

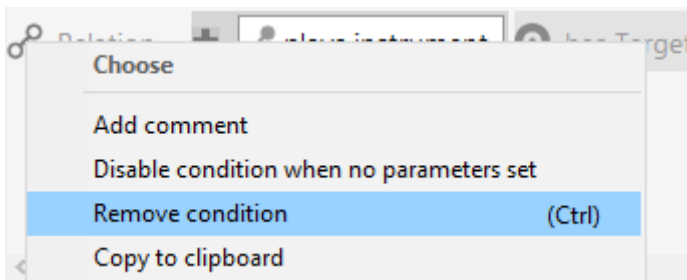
Interaction


If a new structured query is created, the topmost of all types is entered at first per default. In order to limit the query even more you can simply overwrite the name or select "Choose type" by clicking on the icon.

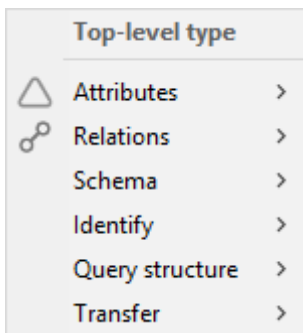


The button  allows you to add more conditions to the structured query. Deleting conditions takes place at the beginning of each line where the type condition is listed (relation, attribute, target, etc.).

Shortcut: 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.



From all possible conditions, focus has, until now, been on the very first item in the menu. A complete explanation of all conditions and options of the structured queries can be found in the next chapters.

1.3.1.1 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.

Example scenario: 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.

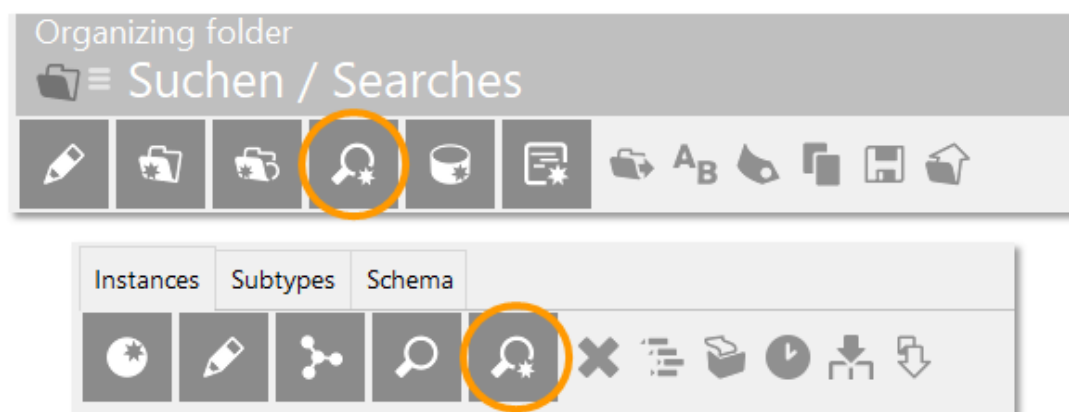
Last but not least, structured queries are also important tools for us as knowledge engineers. We can use them to get an overview of the Knowledge Graph and compile **reports** and **to-do lists**. Here are some examples of questions that can be answered using structured queries:

- 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**.

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.

1.3.1.2 Structure of structured queries

Very indirect conditions can be expressed within structured queries: you may randomly 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.

1.3.1.2.1 Serveral conditions

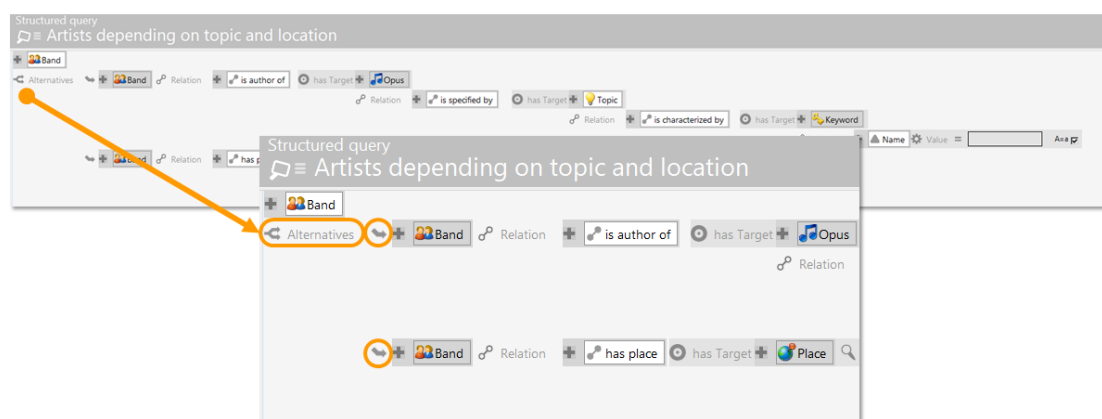
Condition chains may either be randomly deep or several parallel conditions may be expressed: additional conditions are added to any random condition element as a further branch:



Several conditions: English bands with songs on a certain subject

1.3.1.2.2 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:



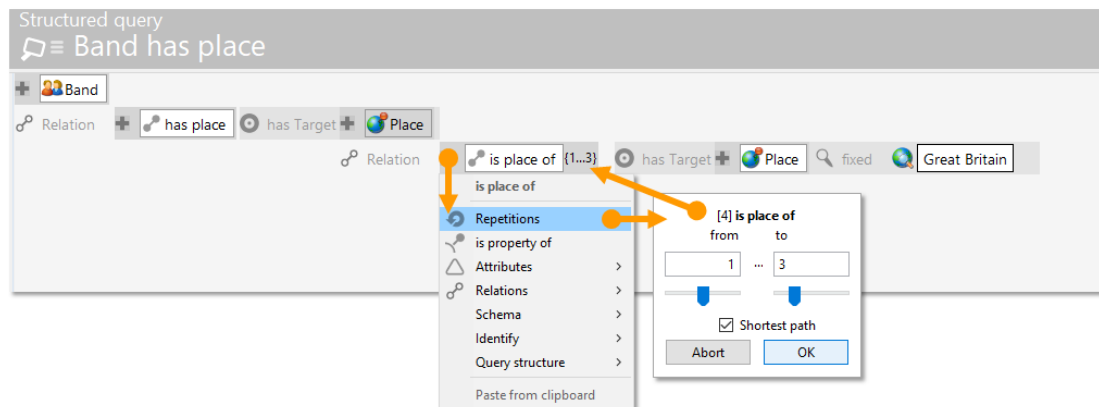
Alternative conditions - the band either has to be English or have songs on a certain subject

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

1.3.1.2.3 Transitivity/Repetitions

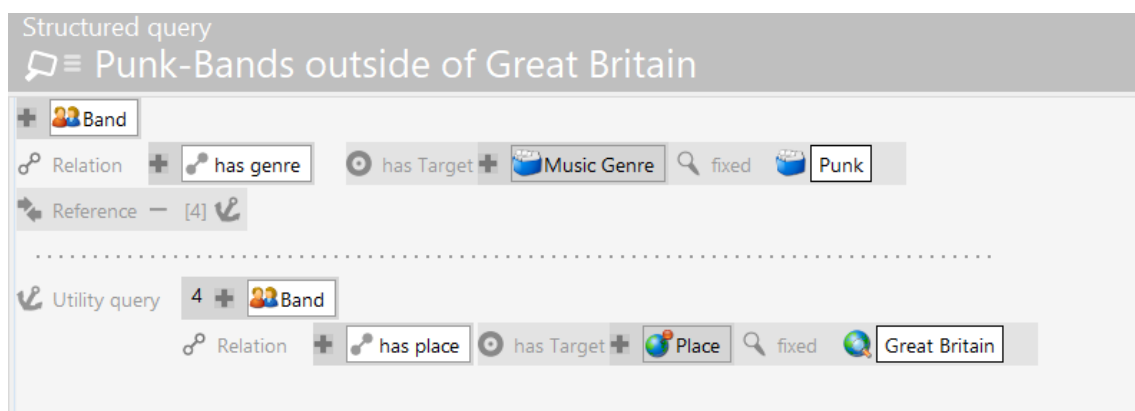
Let's assume the bands are assigned to either cities or countries within the Knowledge Graph. Of these, in turn, it is known which cities are in which countries. In order to document these contents in the search it was possible to very simply expand the condition string: we were able, for example, to 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 does not have to be available.

Simultaneously, we can also include hierarchies which are several levels deep using the function "Repetitions". For example, is known from the band ZZ Top that they come 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 repetitions are reached:



1.3.1.2.4 Negated conditions

Conditions can likewise be purposefully 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 delivers bands from Great Britain - from the main search a reference can be established and thereby noted that the search results are not at all allowed to comply with 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.



Interaction takes place as follows: the utility query is compiled in the type condition and can, after conclusion of the main search above, be linked with the menu item "reference". At this stage you can then select which type the reference should be (in this case negative).

1.3.1.2.5 Corresponds to condition

The reference allows references to be made to other conditions of the same query within a structured 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 (random) bands. Incidentally, the result is, for example, the band "Radiohead" (they covered their own song "Like Spinning Plates").

1.3.1.2.6 Other options in building the structured queries

Structured query macros: Other structured queries but also other searches 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 adapting 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: For all kinds of opus a band can create, albums or songs within an album or songs themselves are taken into account. We need these conditions in forms of partial queries more frequently, for example in a structured query which returns the bands to a certain mood. We start this query with a type condition - we are looking for bands - and integrate the pre-defined module as a condition for these bands:



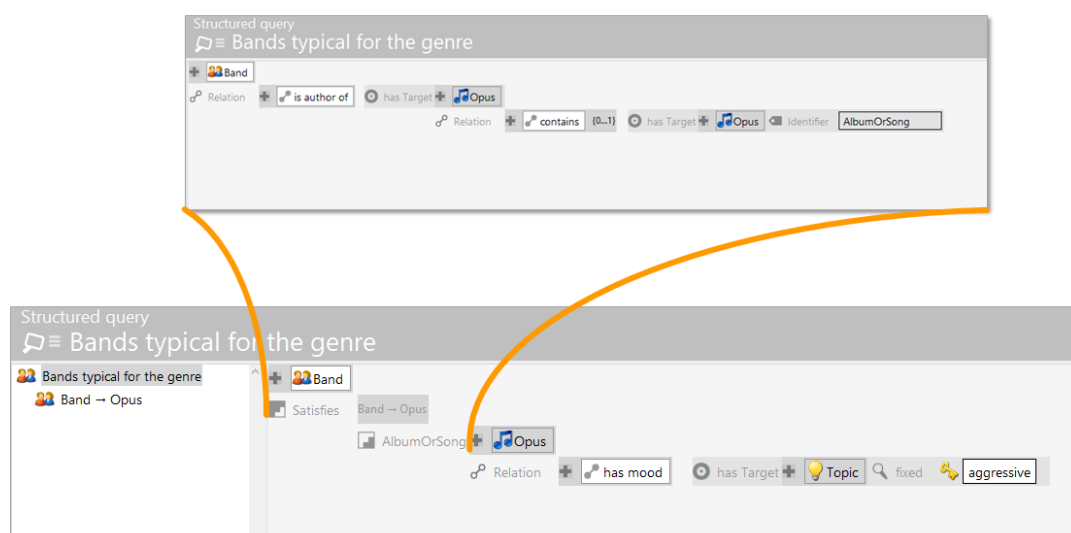
The objects which return those which are integrated into the structured query as macros have, of course, to match the condition with which they are linked from the point of view of their type.

Note: With the aid of the **identifier** function, the query (from the "invoking" 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 selection list with all the registered macros. The advantage is that the macro can be reused for another structured queries.

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

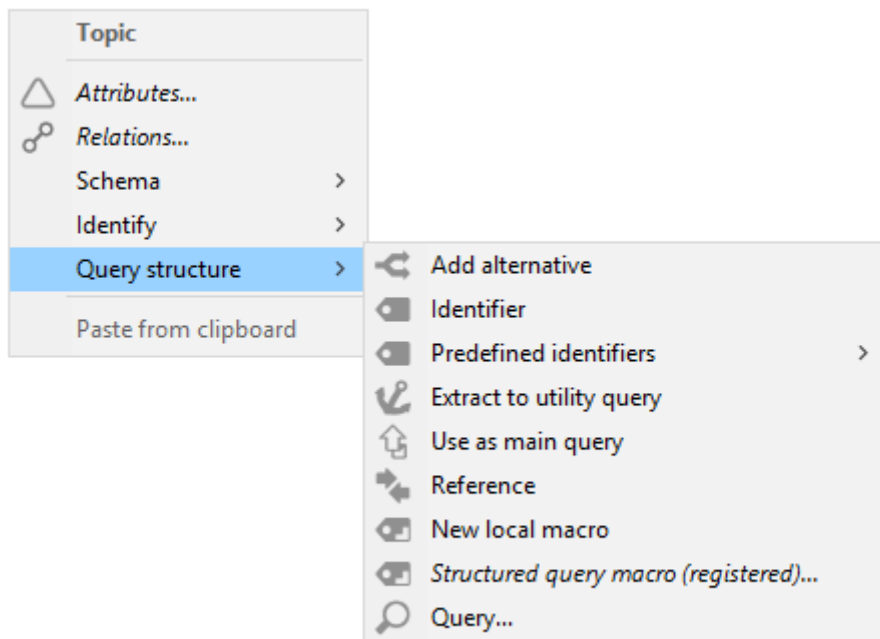
It is also possible to use **local macros** for structured queries. In this case, the macro doesn't get a registry key and is only accessible for and within the respective structured query.



Simple search: Using the search mode "simple search", the results of a simple search or a search pipeline may serve as input for a structured query. Each respective simple search can be selected by means of the selection symbol. 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 without its own conditions may be carried out 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.

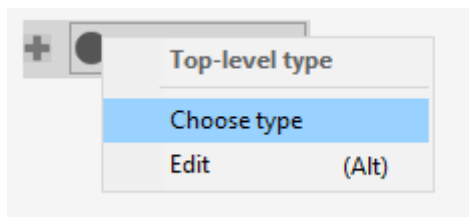
We have thus covered everything we can find within the menu "Query structure":



1.3.1.3 Details of the conditions

The type condition

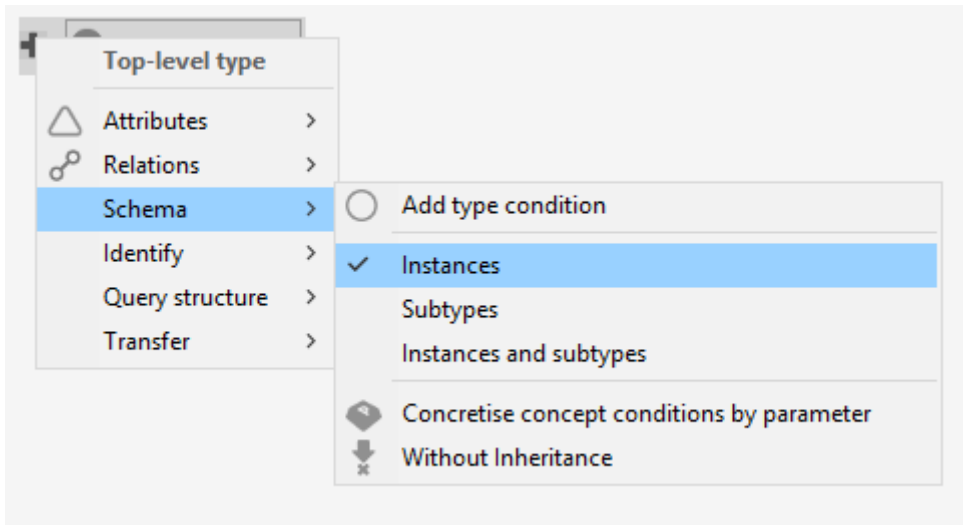
The beginning of the structured query determines which objects should appear as the results. To do so, click on the type icon for the first condition and select "Choose type" in the menu, the input mask then starts in which the name of the object can be entered.



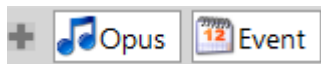
Alternatively, you can simply overwrite the text behind the type icon with the name of the object.

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).

There are further functions available for a type condition. In the item for this there is the item "Schema" in the general condition menu which we can reach via the button +:



Several types of conditions are defined consecutively 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:



We can just search for types of objects instead of specific objects or both at the same time by checking the boxes "Subtypes" and "Instances" in the menu "Schema".

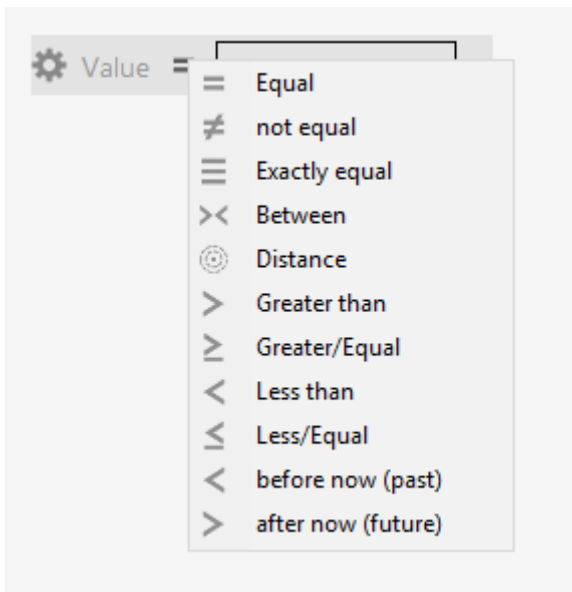


This is what the condition looks like when a search is made for both specific opus as well as subtypes of opus (albums and songs).

Without inheritance: normally, the inheritance starts automatically with 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.

Operators for the comparison of attribute values

Attributes may also play a role as conditions for structured queries. For example, if it does not suffice to only identify objects which show an exact predefined value or the value entered as a parameter. For instance, 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 technical data type of the attribute:

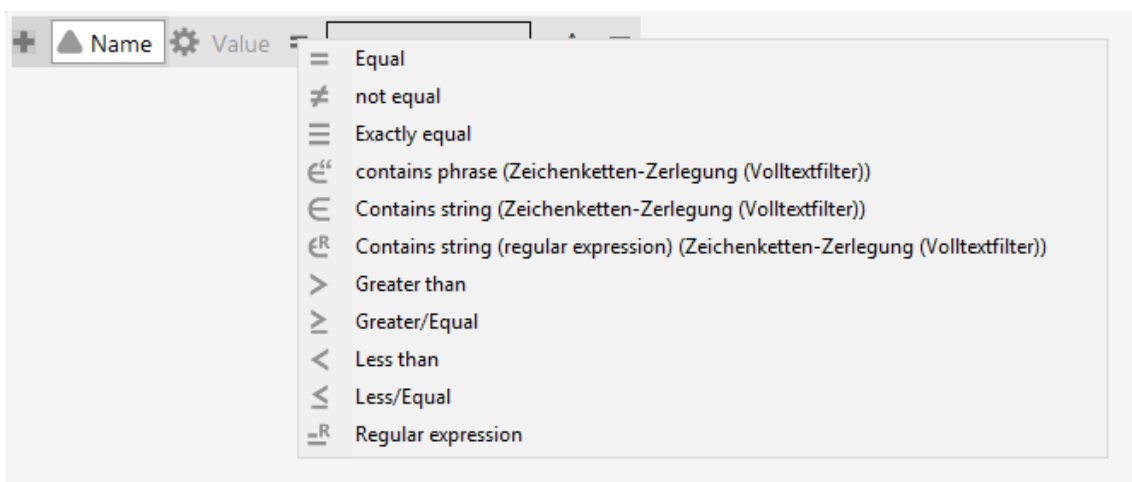


Comparison operators for dates and quantities

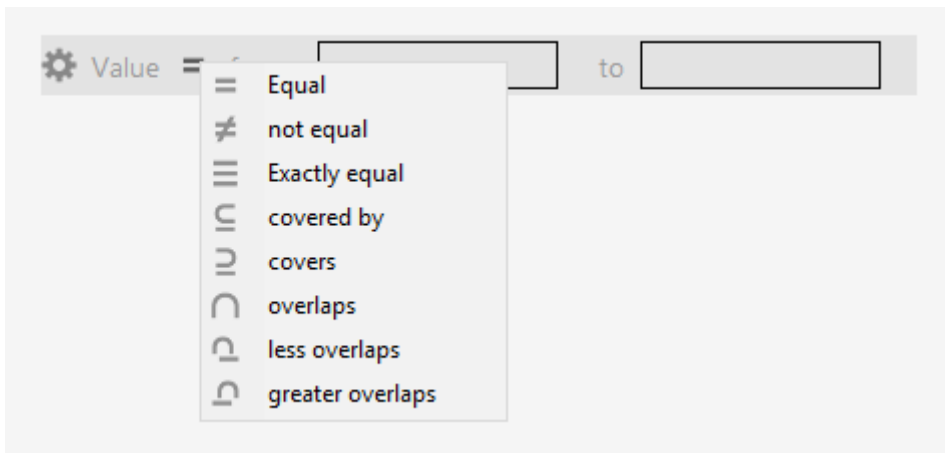
The comparison operator *Exactly equal* constitutes a special case: the index filter is switched off and a search can be made after the special character * which is normally used as a wildcard.

The comparison operator *Between* requires spelling of the parameter value with a hyphen, e.g. "10.1.2005 - 20.1.2005" (interval).

The comparison operator *Distance* requires spelling of the parameter value with a tilde, e.g. "15.1.2005 ~ 5" - i.e. on 15.1.2005 plus/minus 5 days.



Comparison operators for character strings



Comparison operators for intervals

Operator overview

Operator	At-tribute value types	Description	Example
Distance	Date, Geo, Figure	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)	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	Searched value '1 - 5' returns '1 - 3', but not '3 - 6'
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'



Contains character string (regular expression)(strings to word filter)	Character string with full text index	Identifies character strings of which at least one word matches the search terms derived from the regular expression.	Search term 'Ba[yi]e?' finds 'Silke Bayer' and 'Emil Bair', but not 'Bauer'
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'
Equal	Any attribute value type	Identifies 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'
Greater than	All attributes with as-sortable values	Identifies values (and hence the elements carrying the attribute) which are greater than the searched values.	
Greater/Equal	All attributes with as-sortable values	Identifies values greater than or equal to the searched value.	
Less than	All attributes with as-sortable values	Identifies values less than the searched term.	
Less/Equal	All attributes with as-sortable values	Identifies values less than or equal to the searched value.	
before now (past)	Date	Identifies date values that are situated in the past.	



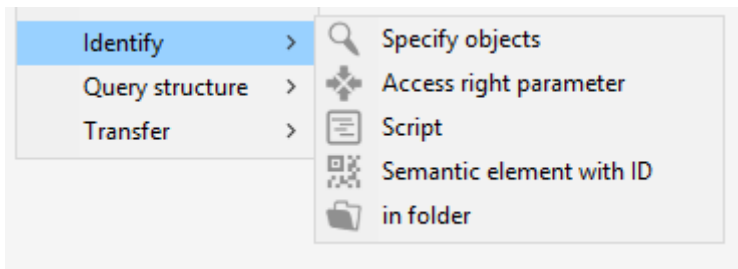
after now (future)	Date	Identifies date values that are situated in the future.	
Regular expression	Character string	Identifies character strings which match the search terms derived from the regular expression.	'\d+\s\w+' finds '64293 Darmstadt'
Covered by	Interval		
Covers	Interval	Identifies intervals which comprise a common, non-empty partial interval with the searched value.	'2 - 4' finds '1 - 3' and '3 - 6', but not '4 - 5'
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'
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.	

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.

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:



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.

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

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.



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 Parameter conditions

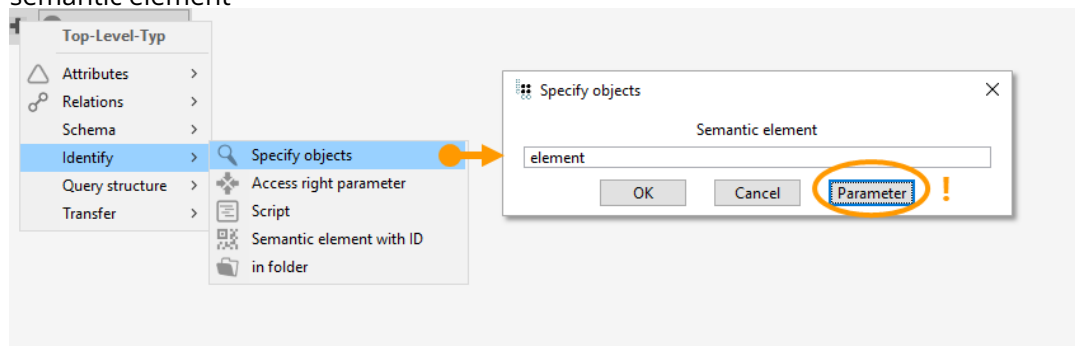
Parameters

In structured queries, input can be passed on by means of a parameter. This allows handing over query input within **JavaScript** code in forms of:

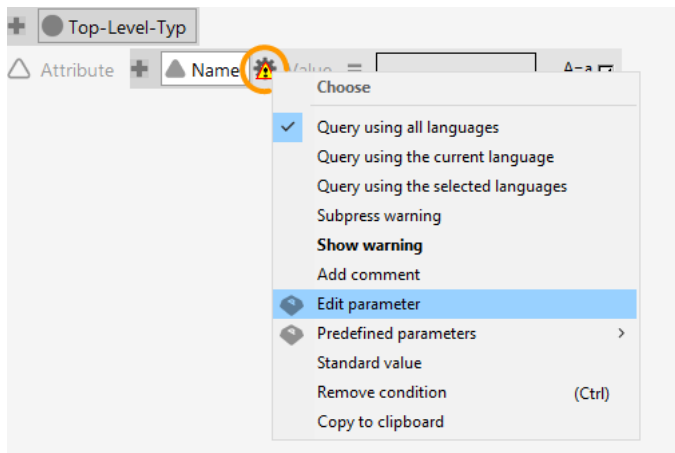
```
$k.Registry.query('<registryKeyOfQuery>').findElements({<parameterNameInQuery>: <input>})
```

The parametrized input can be in forms of:

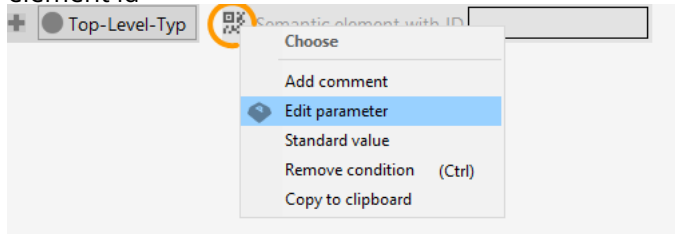
- semantic element



- attribute value



- element id



There are two possibilities to test structured queries using parameters:

1. Using the test environment of the structured query
2. Invoking the structured query by script (executing or debugging)

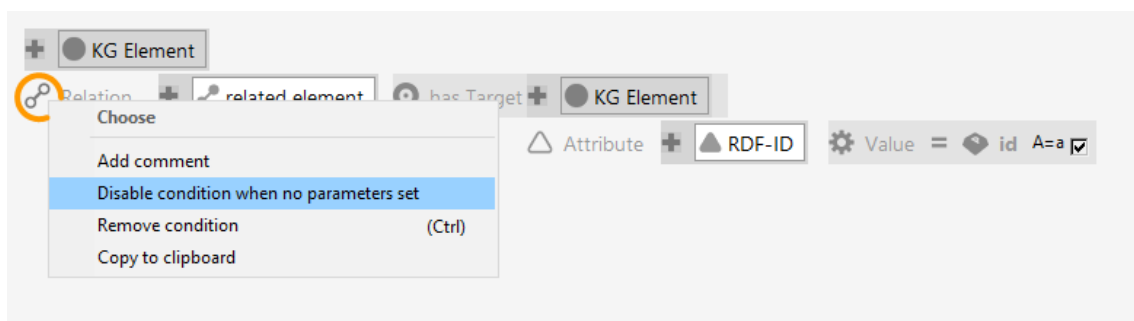
In general, there are four conditions a parameter can have:

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

Optional parameters

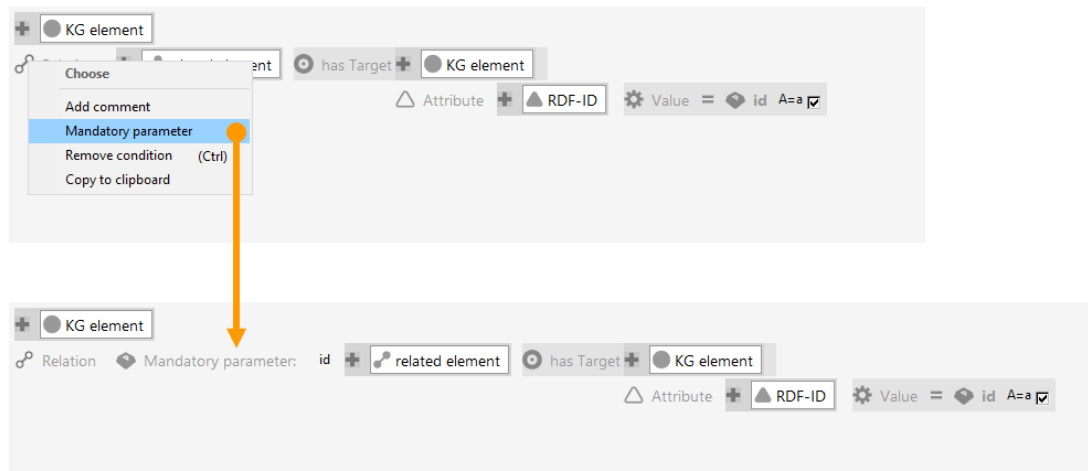
The structured query has a feature that allows using optional parameters: for a certain branch of the query, the context menu offers the condition:

Until 5.4: "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:

Mandatory parameter: element id

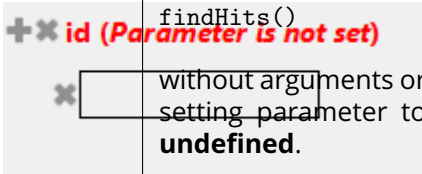
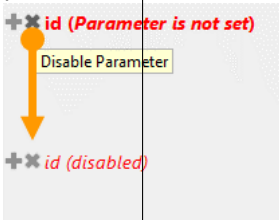
If all mandatory parameters are deactivated, the subsequent query branch will be left out completely when computing the query result, else the parameters which are set are will be used.

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 <code>findElements()</code> or <code>findHits()</code> without arguments or setting parameter to undefined .	Error: "Parameter xy is missing"
Parameter is disabled	Clicking on x besides parameter 	Setting parameter to null .	<ul style="list-style-type: none">• 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 empty 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

Caution: 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).

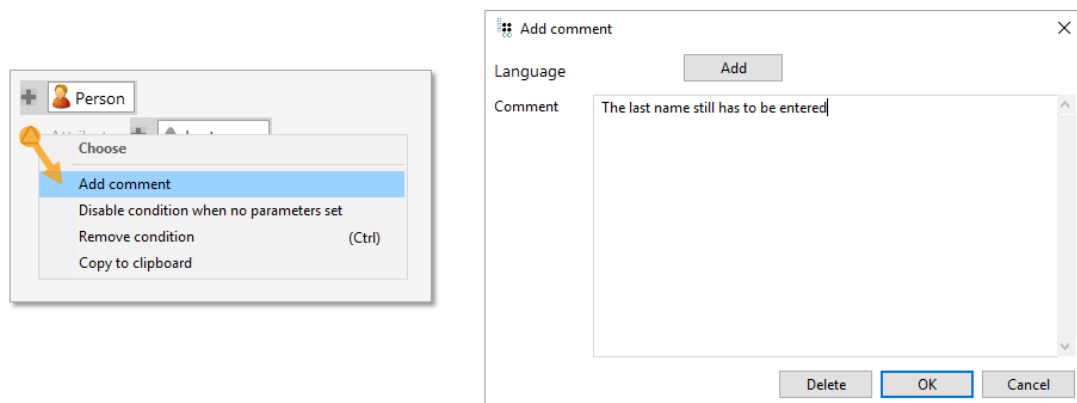
1.3.1.5 Comments in structured queries

Adding comments

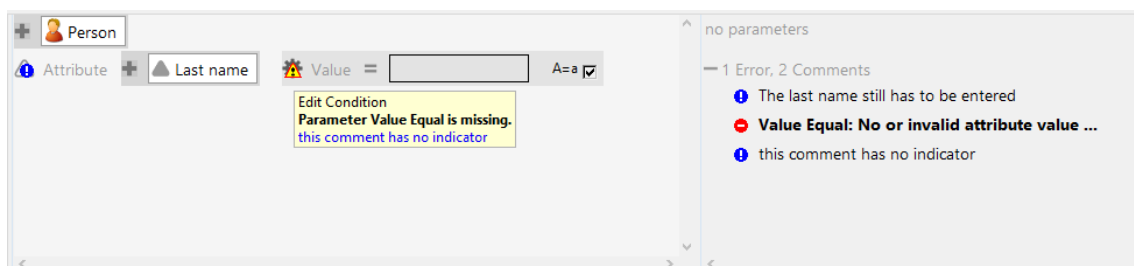
Every condition in a structured query can be commented. For adding a comment, choose the option "add comment" in the context menu. At the condition in the structured query, an existing comment causes a blue indicator flag which shows up a text in case of mouseover.



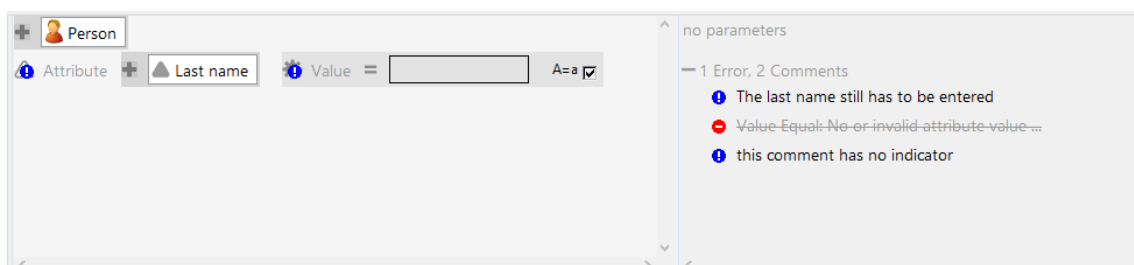
By means of the dialog "Edit comment", the corresponding comment can be changed or removed:



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.



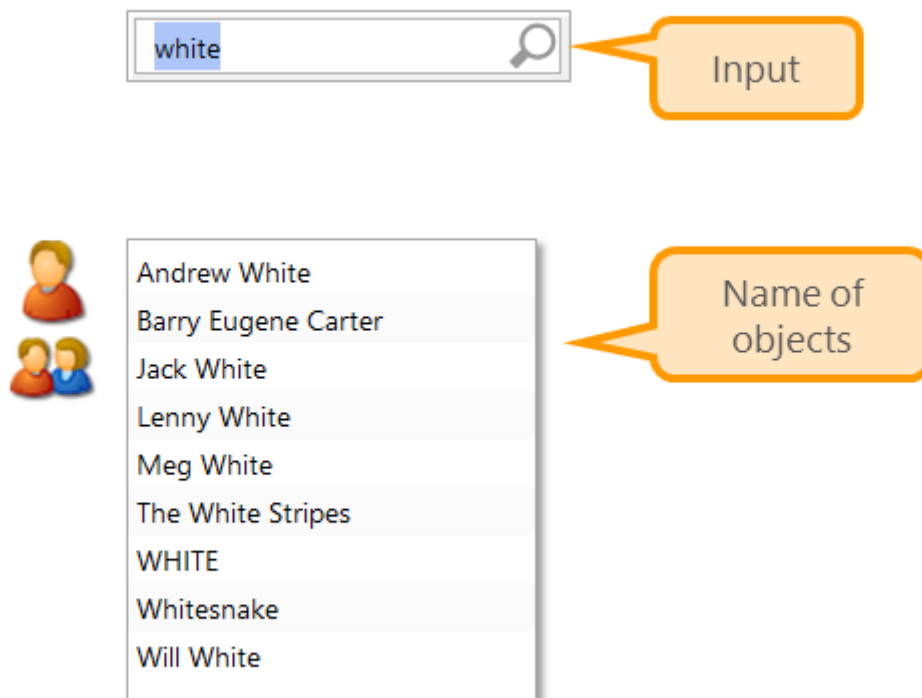
Warnings and cautions can be suppressed in the indicator indication if you want to ignore them at this point (of course, this is not recommended). To do so, click on the indicator symbol in the listed view or choose the function "Suppress warnings" in the context menu of the condition. The indication 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?

- When is the (completely unknown) Chinese experimental band called "WHITE" a hit? If we state that upper case and lower case doesn't matter
- When will we receive "Whitesnake" as a hit? If we understand the entry to be a substring and attach a wildcard
- 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:



Query

Query for artists

Attributes

☐ Hits only on attributes

Filter

- No filter -

Alternative Name

Name Primary name

Semantic elements

☒ filter results

Instances of Band

Instances of Person

Query syntax

☐ Case sensitive

☒ Apply query syntax

☒ Deconstruct query string

Default operator:

OR

Wildcards

☐ No wildcards☐ Auto wildcards☒ Always wildcards

☐ Prefix☒ Substring☐ Suffix

Minimal number of characters

3

Wildcard quality factor

1.0

Language

☒ Query using all languages

☐ Query using the current language

☐ Query using the selected languages

...

Settings

☐ Restrict resultset size

Hits

☐ Server based query

Test environment

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

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			
<input type="radio"/> No wildcards	<input type="radio"/> Prefix	Minimal number of characters	<input type="text" value="3"/>
<input type="radio"/> Auto wildcards	<input checked="" type="radio"/> Substring	Wildcard quality factor	<input type="text" value="1.0"/>
<input checked="" type="radio"/> Always wildcards	<input type="radio"/> Suffix		

- **Placeholder behind** (prefix) finds the [White Lies] for the entry "white"
- **Placeholder in front** (suffix) finds [Jack White]
- **Placeholder behind and in front** (substring) finds [The White Stripes]
- Caution! Placeholder in front is slow.

The option "Always wildcards" works as if we had actually attached an asterisk in front and/or behind. Behind automatic 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 weighting factor for wildcards 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, him/herself use placeholders in the search entry and these can be included in the search.

Apply query syntax: when the box for the option "Apply query syntax" has been checked a simplified form of the analysis of the search input is used in which, for example, the words "and" and "or" and "not" no longer have a steering effect. 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 for attributes" has been set). A hit for online AND system thus delivers semantic objects which have a matching attribute for both online and the system (which is not necessarily the same).



Filtering: simple searches, full-text searches and also some of the specialised searches may be filtered according to the types of objects. In the example described in the last paragraph 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.

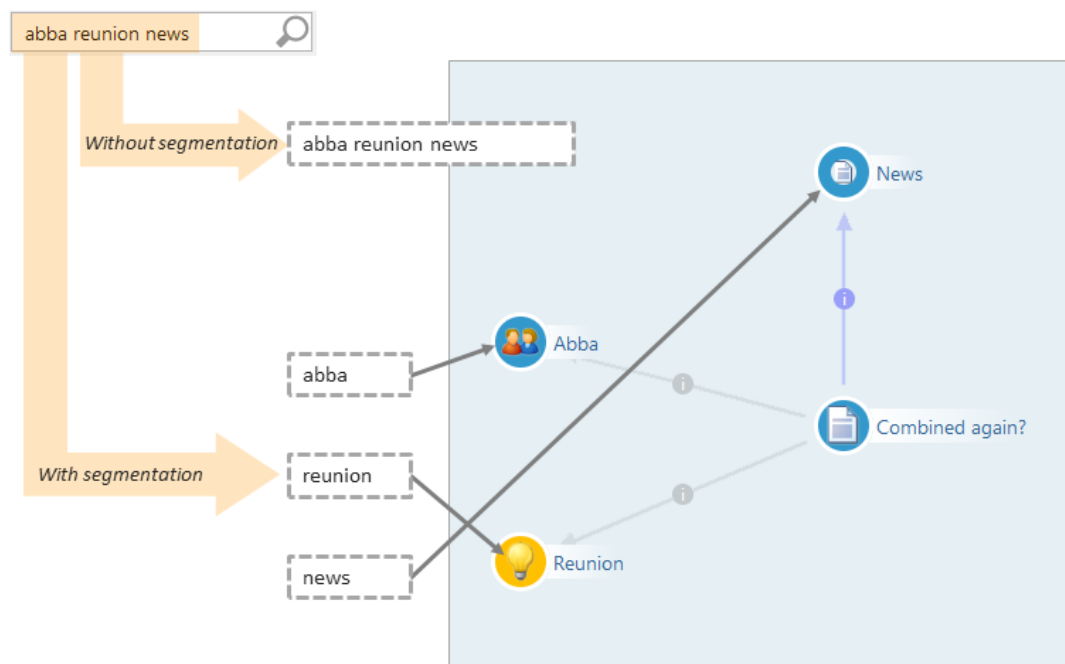
Translated attributes: in the case of translated attributes we can neither select a translation, nor have the language dynamically defined. Search for multilingual attributes, then in the active language or in all languages, depending on whether the option "in all languages" is checked.

Query output: a maximum query output may be defined by entering the maximum number in the "results" box. This checkbox will then limit the query output and the mechanism can be activated or deactivated. By entering the number in the output the checkbox will automatically be activated. Caution: if the number is exceeded no output will be shown!

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 be 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 categorised 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 Fulltext search and indexation

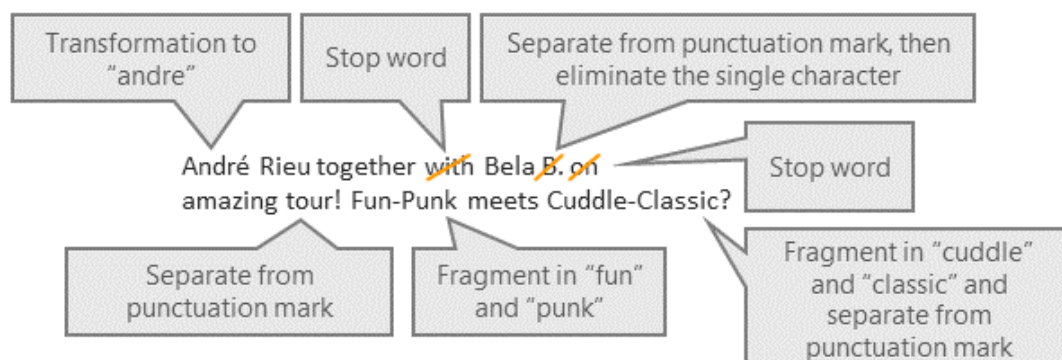
If we want to view or search through longer texts word by word, e.g. description attributes we recommended the use of full-text index. What does something like that look like?

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:



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 "Cuddle-Classic" 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 {cuddle} and {classic}. In the first instance our message will then only be found if an exact search is made for "Cuddle-Classic" or, for example, "*uddle-c*", in the second instance for all "classic" 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.

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. Known: **stop words**, at this point we can maintain a list. Moreover, we probably do not want **individual words** (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 organised 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.

Example

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



Step 1 of our search pipeline goes from a starting mood (in this case "aggressive") via the relation *is mood of* to the songs which are assigned to the mood 'aggressive':

Search Pipeline
typical bands

Components

typical bands

- by songs
 - Weighted relation/attribute (is mood of) mood => songs

Configuration Hits Cause Description

Input mood Hit

Output songs Hit

Properties is mood of (Instances of Opus) Add Remove

Weight Remove ...

Standard value 0.25

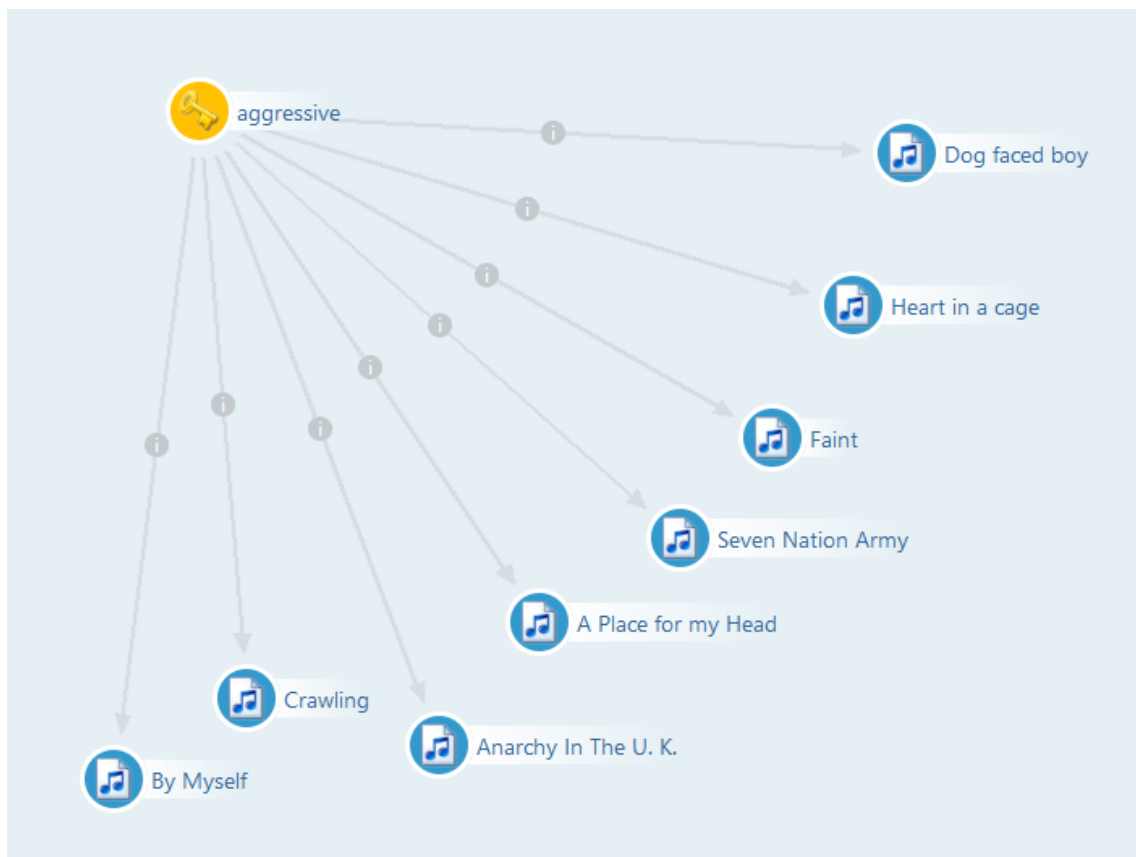
Add Remove Move up Move down

Settings

☐ Restrict resultset size Hits

☐ Server based query

Test environment



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*:



Search Pipeline
🔍 typical bands

Components

🔍 typical bands

- by songs
 - Weighted relation/attribute (is mood of) mood => songs
 - Weighted relation/attribute (has author) songs => bandsBySongs

Configuration Hits Cause Description

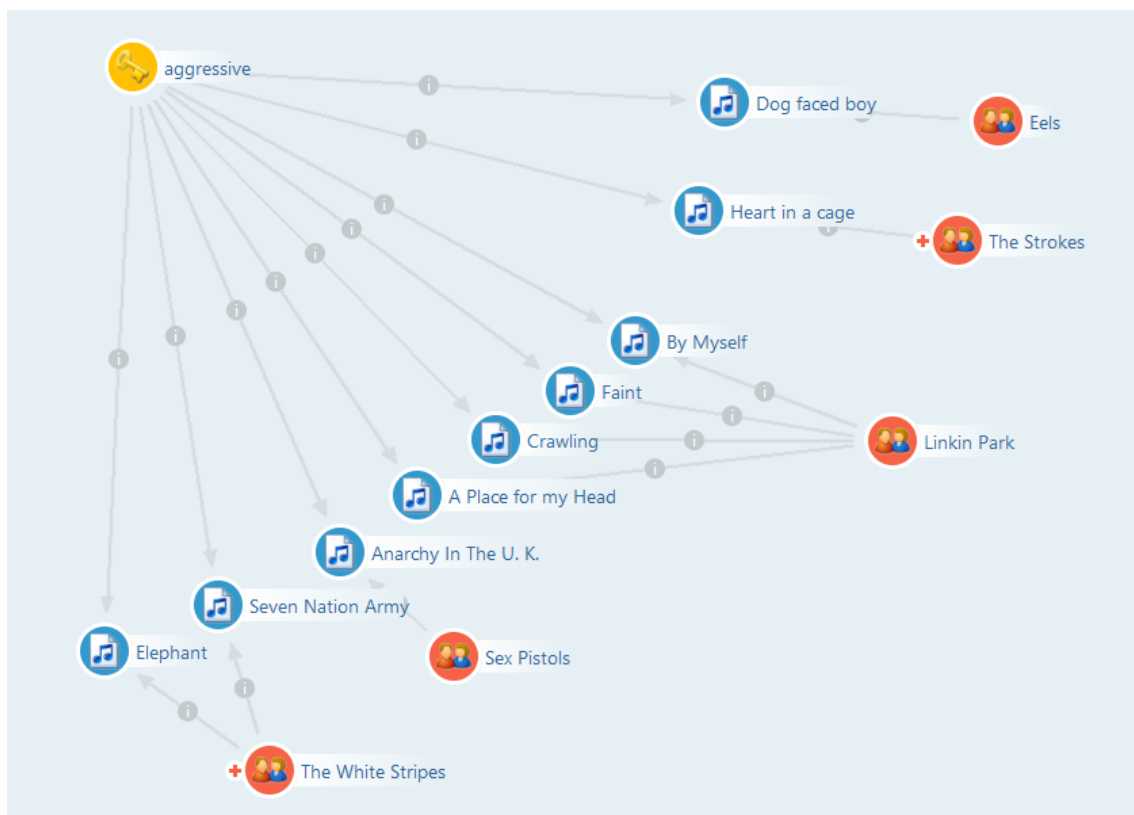
Input songs Hit

Output bandsBySongs Hit

Properties has author (Instances of Band) Add Remove

Weight Remove ...

Standard value 1.0



Now we would like to pursue a second path: from the starting point 'mood' "aggressive" to the musical directions which are characterised 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:



Search Pipeline
= typical bands

Components

- typical bands
 - by songs
 - Weighted relation/attribute (is mood of) mood => songs
 - Weighted relation/attribute (has author) songs => bandsBySongs
 - by style
 - Structured query "Band" => bandsByStyle
 - Scale quality bandsByStyle => bandsByStyle
 - Merge hits bandsBySongs, bandsByStyle => typicalBands
 - Result typicalBands

Configuration Query parameters Description

Input Hits to filter, or no input to perform the query

Output bandsByStyle

Remain Hits

Hits that do not match the query condition

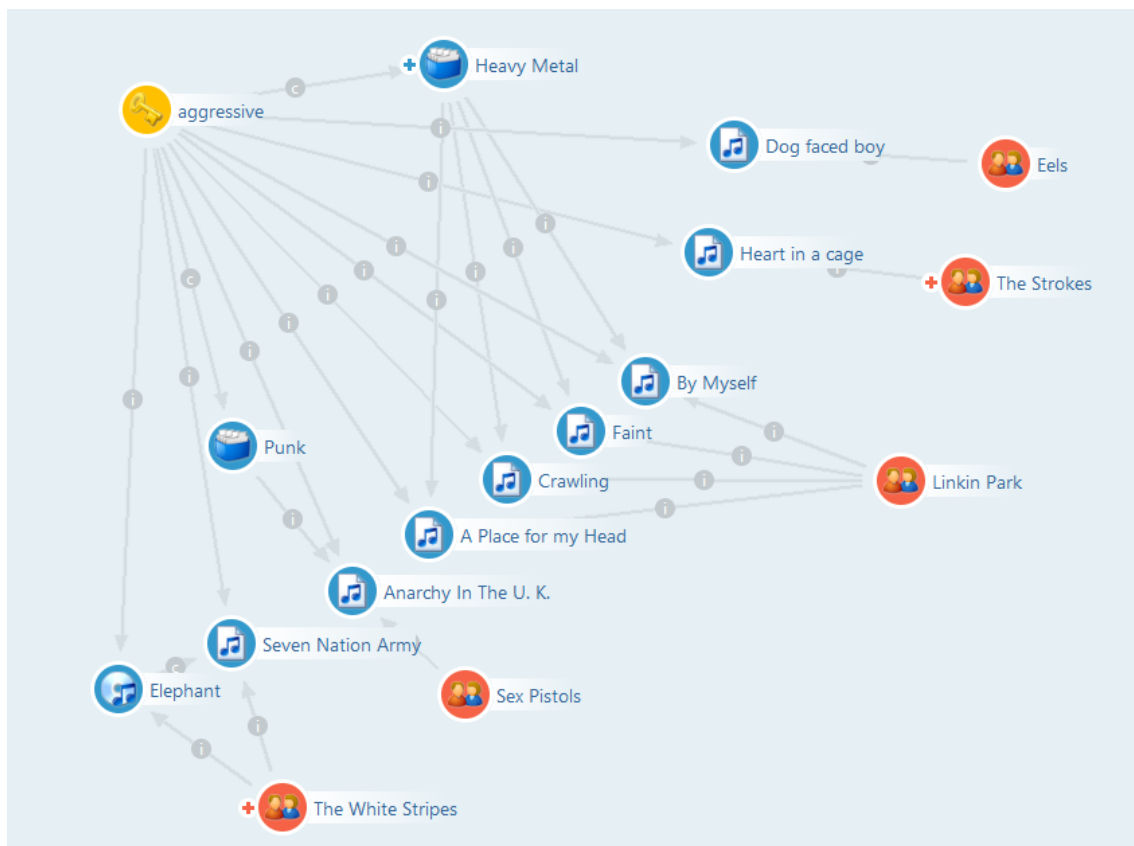
Query Structured query Open

Band

Relation has style has Target Style

Relation is characterized by has Target Keyword

Attribute Name Value mood Area



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



Search string

Parameters

Name	Required	Type	Value	Type of value
mood	<input checked="" type="checkbox"/>		aggressive	Keyword







Set value

Set element

Reset

Search

Trace search



Name	Type	Reason	Search string	Quality
Linkin Park	Band	Crawling, By Myself, A		100
The White Stripes	Band	Seven Nation Army, El		95
Sex Pistols	Band	Anarchy In The U. K.		80
Eels	Band	Dog faced boy		78
The Strokes	Band	Heart in a cage		78

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.

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.

1.3.3.1 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.

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":

Search Pipeline
typical bands

Components

- typical bands
 - by songs
 - Weighted relation/attribute (is mood of) mood => songs
 - Weighted relation/attribute (has author) songs => bandsBySongs

Configuration Hits Cause Description

Input: songs

Hit

Output: bandsBySongs

Hit

Properties: has author (Instances of Band)

Add Remove

Weight:

Remove ...

Standard value: 0.7

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.

Search Pipeline
typical bands

Components

- typical bands
 - by songs
 - Weighted relation/attribute (is mood of) mood => songs
 - Weighted relation/attribute (has author) songs => bandsBySongs
 - by style
 - Structured query "Band" => bandsByStyle
 - Scale quality bandsByStyle => bandsByStyle
 - Merge hits bandsBySongs, bandsByStyle => typicalBands
 - Result typicalBands

Configuration Description

☐ Add hit causes

Parameters

Name	Required	Type	Description
mood	<input checked="" type="checkbox"/>		

Some components enable a deviation from the standard processing sequence:

Individual processing: elements of a quantity, 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.

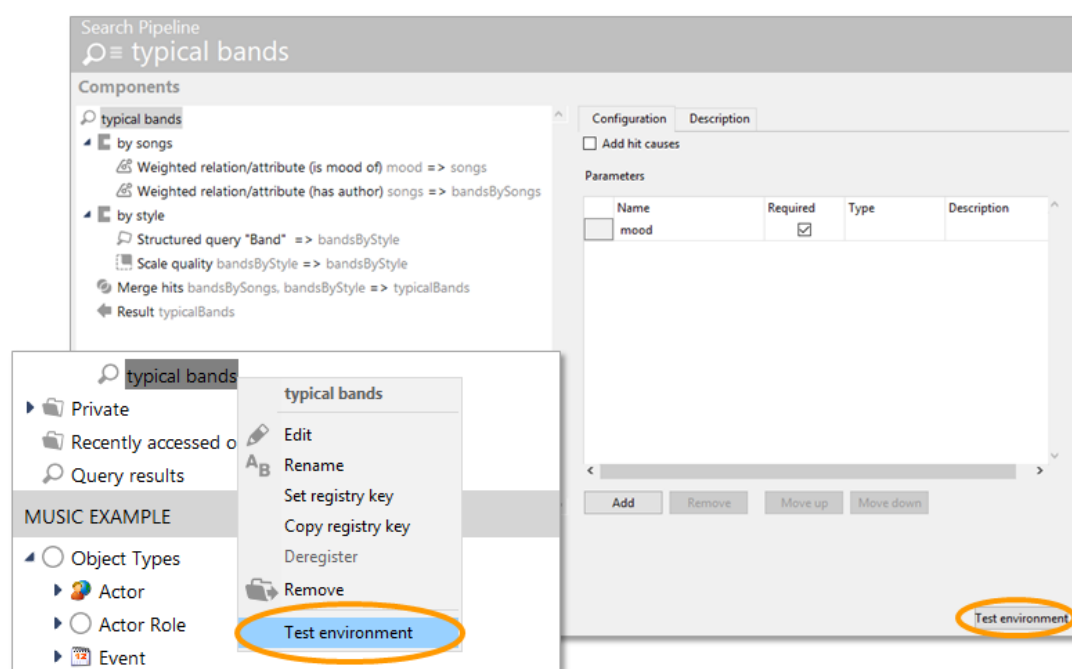
KPath condition: By using a KPath condition we can determine that the sub components may only then be implemented if a condition expressed in KPath is fulfilled. If the condition is not fulfilled the input will be adopted. KPath is described in the manual for KScript.

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

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.

Test environment

The test environment can be invoked by several ways:



Using the test environment in the menu we can analyse 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.



Search string

...

Parameters

Name	Required	Type	Value	Type of value
mood	<input checked="" type="checkbox"/>		aggressive	Keyword

Set value

Set element

Reset

Search

Trace search

Name	Type	Reason	Search string	Quality
Linkin Park	Band	Crawling, By Myself, A	.	100
The White Stripes	Band	Seven Nation Army, El	.	95
Sex Pistols	Band	Anarchy In The U. K.	.	80
Eels	Band	Dog faced boy	.	78
The Strokes	Band	Heart in a cage	.	78

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.

Trace search

by songs

Weighted relation/attribute (is mood of) mood => songs

Weighted relation/attribute (has author) songs => bandsBySongs

by style

Structured query "Band" => bandsByStyle

Scale quality bandsByStyle => bandsByStyle

Merge hits bandsBySongs, bandsByStyle => typicalBands

Result typicalBands

Duration: 3.03 milliseconds

Messages:

Variables and values

Name	Type	Value
songs	Output	(9) Dog faced boy, Elephant, A F
mood	Input	aggressive

Hits

Name	Type	Reason	Search string	Quality
A Place for my He	Song	.	.	25
Anarchy In The U	Song	.	.	25
By Myself	Song	.	.	25
Crawling	Song	.	.	25

Calculation possibilities

In the case of some components it is possible to summarise several quality values into one single quality value - e.g. in "summarise 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



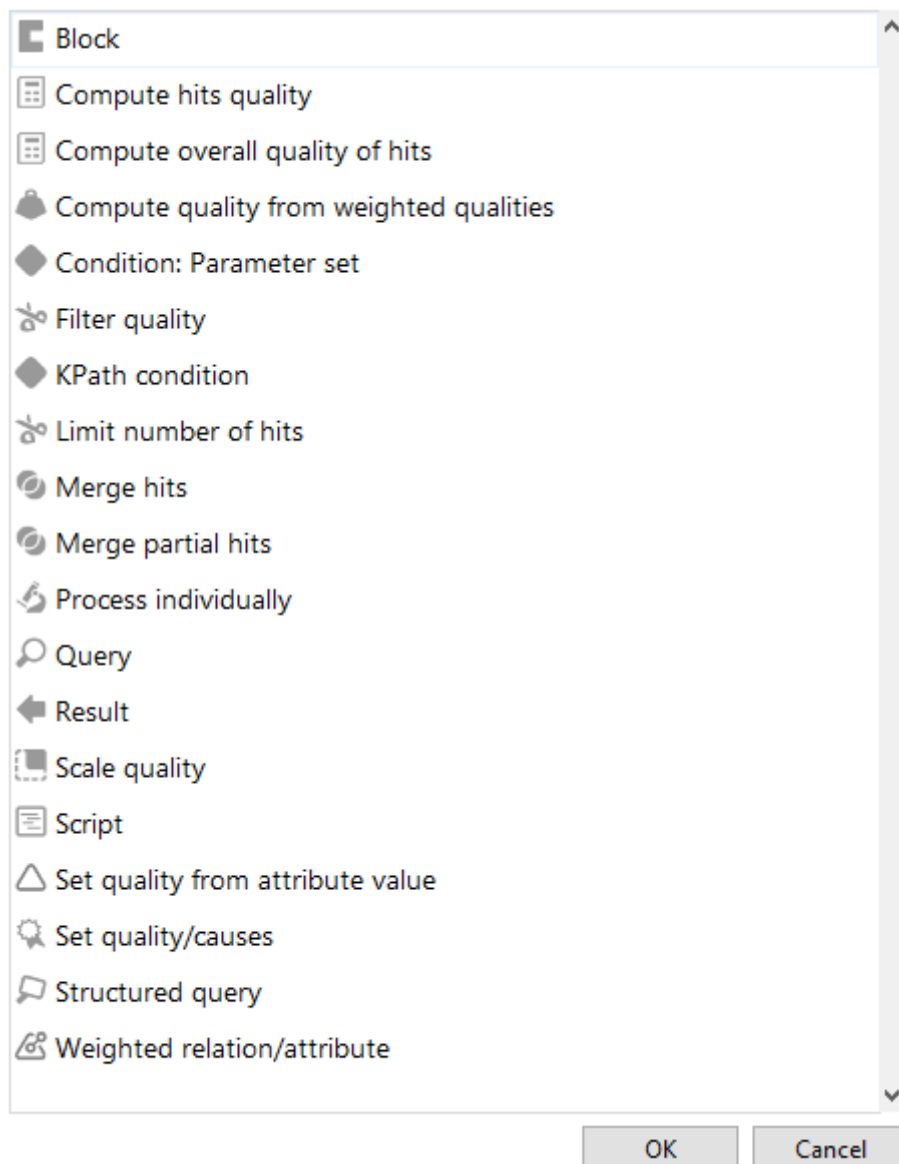
- ranking

The option "ranking" is then always 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%.

1.3.3.2 The single 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:



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.

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.

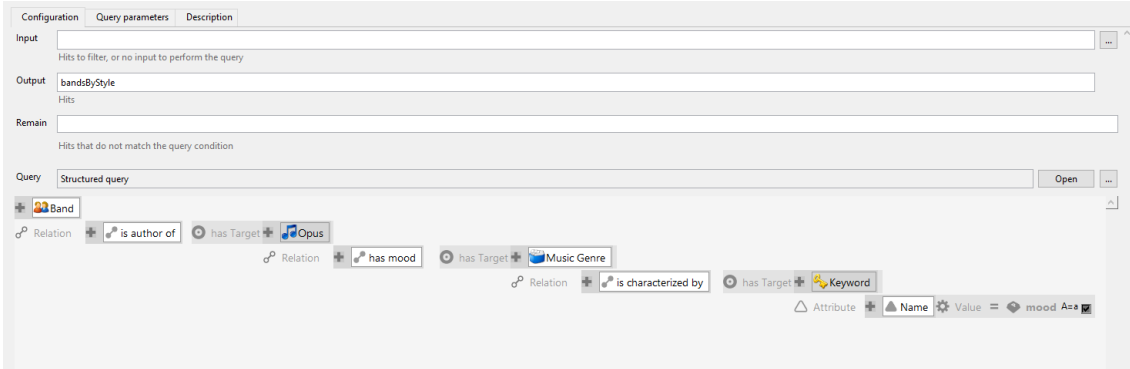
Instead of assigning a fixed weight for moving along the relation, we could also read the value from a meta-property of the basic type float of the selected relation. If the attribute is not available or no attribute has been configured, the default value is used. The value should be between 0 and 1. 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.

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.

Please 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.

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 ("hit collections") 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.

Merge hits

We can use this component to summarize different sets of hits ("hit collections") 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 ("hit collections") 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 ("hit collections") 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.

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.

Process individually

This element is used for processing several hits (= array of hits) in order to use each single hit (= n^{th} 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.



Example: 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.

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".

Script

A search pipeline can contain a script (JavaScript or KScript). 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 and KScript are described in separate manuals.

Set quality/causes

For hits arising as a result caused by the input from preceeding (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 (preceeding) 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.

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.



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.

Compute overall quality of hits

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

Filter quality

We can restrict sets of hits ("hit collections") 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.

Limit number of hits

If the total number of a set of hits is to be restricted, we can add the component "restrict 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.

If 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.

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.

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.

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.

1.3.3.3 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, ...).

Examples:

- **@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

1.3.3.3.1 Names

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

Name	Description
<i>name</i>	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: [a-zA-Z_]*[^\(\)\{\}\\$%{}[],~@\$#+-'"s ^\&]* (For better reading, the escape character "\" has been left out)
"name" 'name'	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 "\" serves as escape character for possibly used apostrophes, e. g. 'Wendy\'s'.



<i>name</i> {lang}	Name in the specified language "lang"
<i>\$name</i> \$, <i>\$"name"</i> \$	Internal name
<i>\$name</i> §, <i>§"name"</i> §	System name
#ID42_1013	ID of the object

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

1.3.3.3.2 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.

Example: 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):

Opera- tor	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
^^	xor	Exclusive or (logical operator)
&&	and	And (logical operator)
	or	Or (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)"/>
```

1.3.3.3 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.

Example: Name of all books which had been written by Goethe:

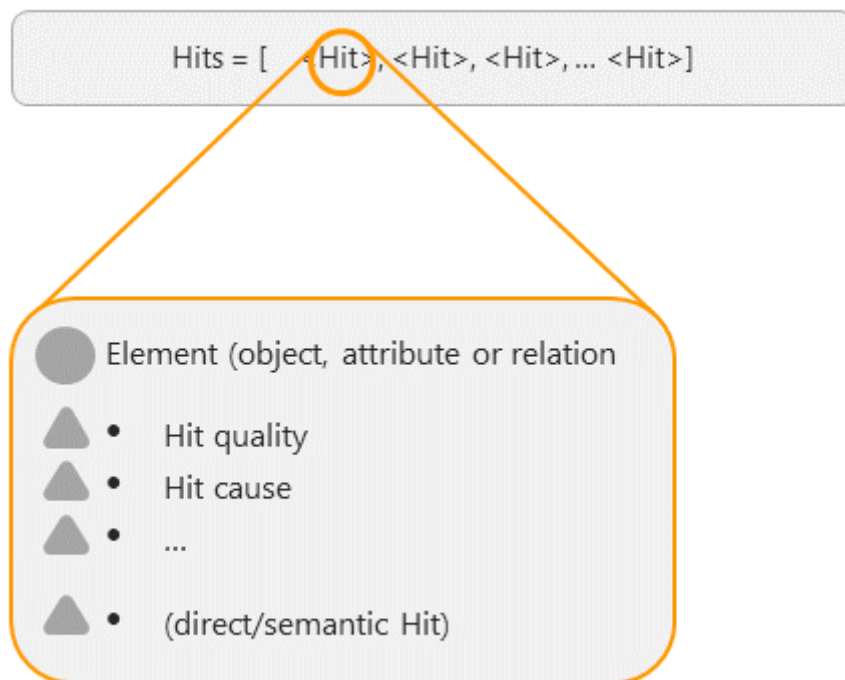
```
//book\[~author/target()/@Name = "Goethe"]/@Name/value()
```

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:



Meta-attributes of hits

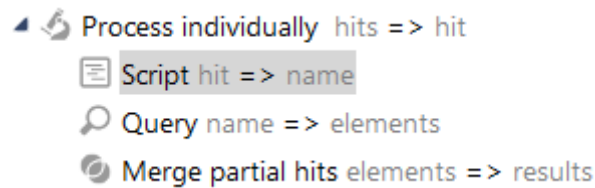
In addition to the semantic element, the following meta-attributes are transported in a hit:

- **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.

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.



Example script for converting a hit into a string:

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

Using hits in tables

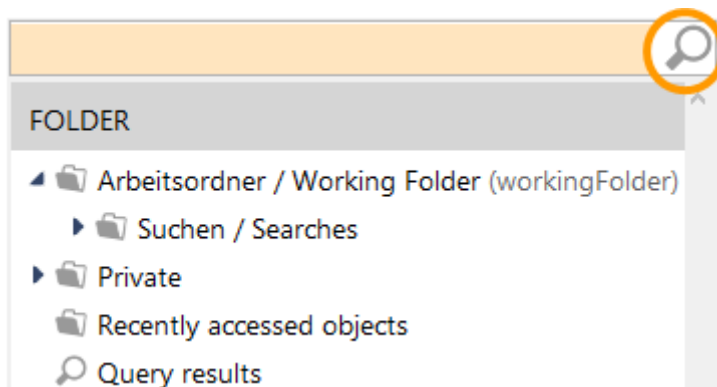
The “Use hits” option is available in the column element configuration of a table. This option determines whether the entire hit element (semantic element + meta-attributes) or only the semantic element is to be forwarded to display query results.

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 The 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 have to drag & drop a pre-configured search only 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 contains the search mode which was last carried out.

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 Lucene

The full-text search may also alternatively be carried out via the external indexer Lucene. The search configuration is then analogue 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 analogue. 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



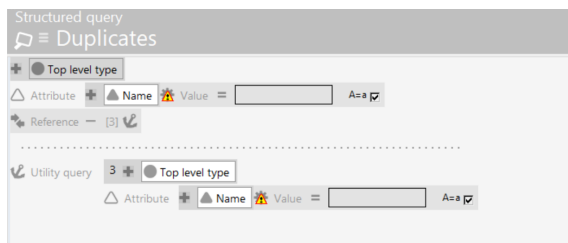
used.

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 an unusually amount of time. It therefore is advised to restrict the query to the most exactly subtype as possible.

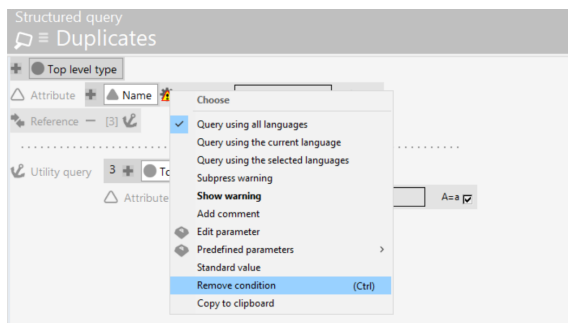
In principle, the structured query searches for **different** objects that have identical values for their identifying attributes (here: objects with identical names).

The query for duplicates can be configured as follows:

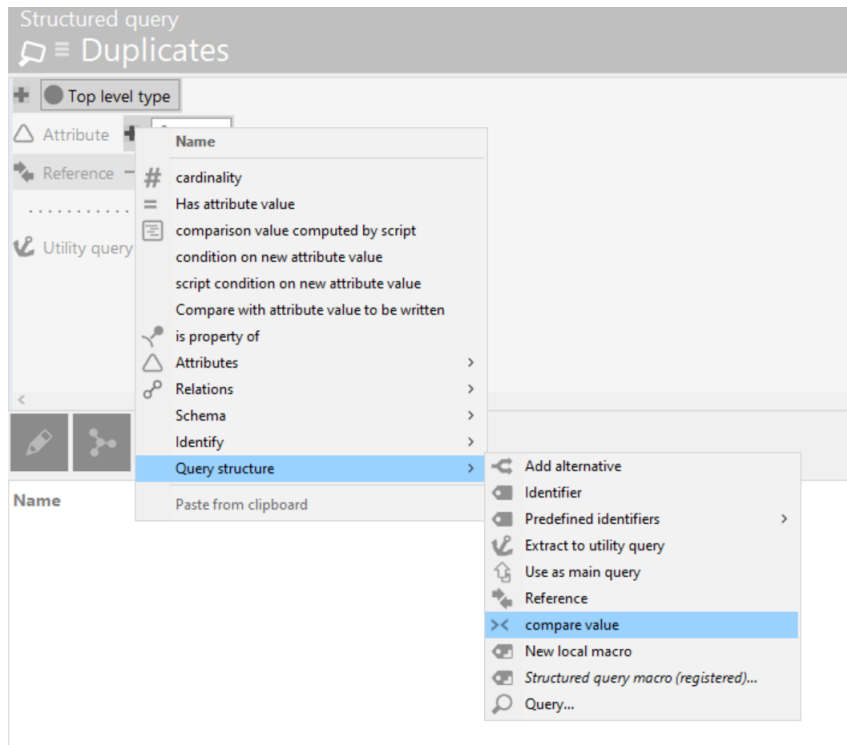
1. Create a query for objects of the subtype in question. Add the identifying attribute as condition (here: primary name).
2. Depending on the object, create a utility query. Use a negative reference (comparison operator "is not") to make sure that only different objects will be found:



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.

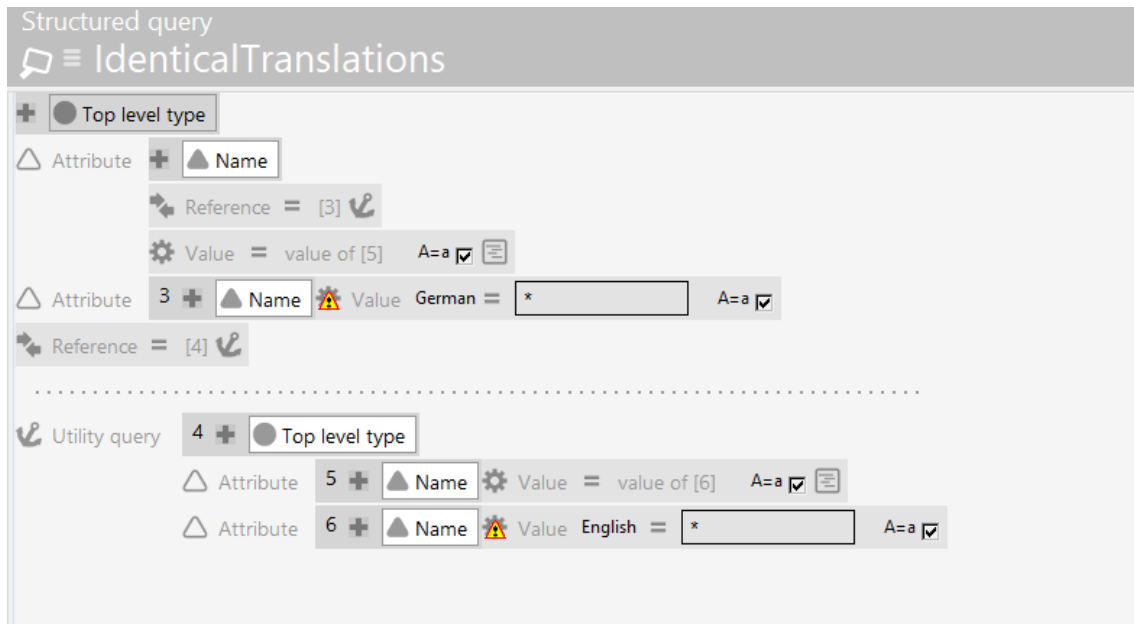


1.3.6.5 Query for identical translations

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 an unusually amount of time. It therefore is advised to restrict the query to the most exactly subtype as 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:



1.4 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&Drop, and there are also operations to, for example, define result lists in folders.

In 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.


Caution: 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  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

Shift, 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.

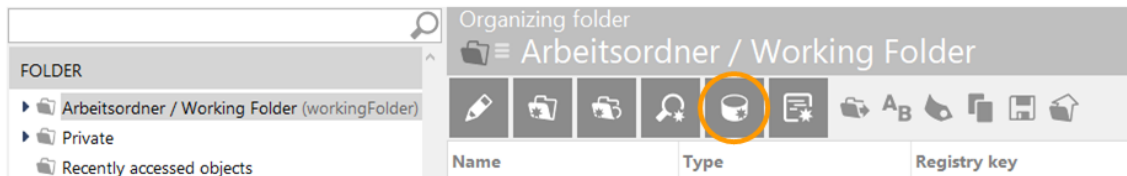
Folder settings

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. This, at least, until the second limit has been reached: Maximum number of query outputs, maximum number of outputs in object lists - in this instance high values - there is actually no result when these values are exceeded, queries must be restricted, e.g. in object lists in which we also have the beginning of the name in the input box.

1.5 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.



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/Excel file
- XML file
- MySQL interface
- ODBC interface
- Oracle interface
- PostgreSQL interface
- For the exchange of user IDs, a standard LDAP interface has been implemented.

The following section uses a CSV file to describe how to create a table-oriented import/export. As all imports/exports apart from XML imports/exports are table-oriented and the individual data sources differ only in terms of their configuration, the example for the mapping of the CSV file can also be applied to the mapping of other databases and file formats.

1.5.1 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.

1.5.1.1 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:



	A	B	C	D	E	F	G	H
1	Title name	Artist	Album	Genre	Run-time	Year	My rating	
2	The suburbs	Arcade Fire	The suburbs	Postwave	315	2010	60	
3	Ready To Start	Arcade Fire	The suburbs	Postwave	255	2010	80	
4	Modern Man	Arcade Fire	The suburbs	Postwave	279	2010	60	
5	Rococo	Arcade Fire	The suburbs	Postwave	236	2010	40	
6	Empty Room	Arcade Fire	The suburbs	Postwave	171	2010	20	
7	City With No Children	Arcade Fire	The suburbs	Postwave	191	2010	20	
8	Half Light I	Arcade Fire	The suburbs	Postwave	253	2010	20	
9	Half Light II (No Celebration)	Arcade Fire	The suburbs	Postwave	267	2010	40	
10	Suburban War	Arcade Fire	The suburbs	Postwave	281	2010	80	
11	Month Of May	Arcade Fire	The suburbs	Postwave	230	2010	20	
12	Wasted Hours	Arcade Fire	The suburbs	Postwave	200	2010	40	
13	Deep Blue	Arcade Fire	The suburbs	Postwave	268	2010	60	
14	We Used To Wait	Arcade Fire	The suburbs	Postwave	301	2010	100	
15	Sprawl I (Flatland)	Arcade Fire	The suburbs	Postwave	174	2010	40	
16	Sprawl II (Mountains Beyond Mountains)	Arcade Fire	The suburbs	Postwave	318	2010	40	
17	The Suburbs (Continued)	Arcade Fire	The suburbs	Postwave	87	2010	40	
18	Eleanor Rigby	The Beatles	Revolver	Oldies	127	1966	60	
19	For No One	The Beatles	Revolver	Oldies	121	1966	60	
20	Good Day Sunshine	The Beatles	Revolver	Oldies	129	1966	40	
21	Here There And Everywhere	The Beatles	Revolver	Oldies	145	1966	40	
22	I Want To Tell You	The Beatles	Revolver	Oldies	149	1966	40	
23	I'm Only Sleeping	The Beatles	Revolver	Oldies	181	1966	60	
24	Love To You	The Beatles	Revolver	Oldies	181	1966	20	
25	She Said She Said	The Beatles	Revolver	Oldies	157	1966	40	
26	Taxman	The Beatles	Revolver	Oldies	159	1966	20	
27	Tomorrow Never Knows	The Beatles	Revolver	Oldies	177	1966	20	
28	Yellow Submarine	The Beatles	Revolver	Oldies	160	1966	20	
29	About A Girl	Nirvana	MTV Unplugged in NY	Rock	217	1994	60	
30	Jesus Doesn't Want Me For A Su	Nirvana	MTV Unplugged in NY	Rock	277	1994	40	
31	The Man Who Sold The World	Nirvana	MTV Unplugged in NY	Rock	260	1994	80	
32	Pennyroyal Tea	Nirvana	MTV Unplugged in NY	Rock	220	1994	60	
33	Dumb	Nirvana	MTV Unplugged in NY	Rock	172	1994	40	
34	Polly	Nirvana	MTV Unplugged in NY	Rock	196	1994	60	

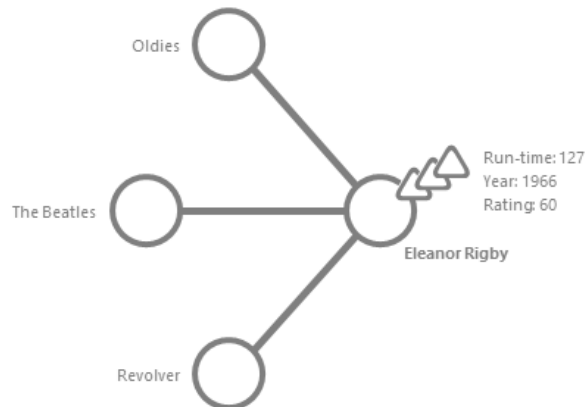
Using the song as a basis, we build up the structure of attributes, relations and target objects that should be created by the import (left-hand side). An object of type song is created this way for row 18, for example, with the following attributes and relations:



CSV/Excel file
Mood example

Mood example

- 1: Instances of **Song**
 - 2: Attribute **Name** English
 - 3: Attribute **run-time (seconds)**
 - 4: Attribute **Year**
 - 5: Attribute **Rating**
- 6: Relation **has genre**
 - 7: Instances of **Music Genre**
 - 8: Attribute **Name** English
- 9: Relation **has author**
 - 10: Instances of **Band**
 - 11: Attribute **Name** English
- 12: Relation **is contained by**
 - 13: Instances of **Album**
 - 14: Attribute **Name** English

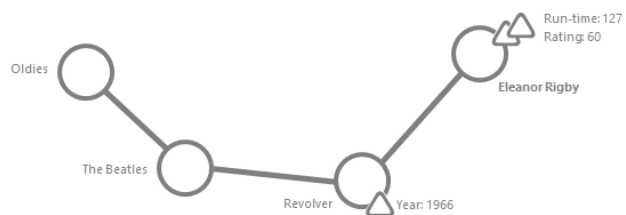


We can, however, also decide to distribute the information from the table in a different way, for example allocate the year of release and artist to the album, and in turn the genre to the artist. A row still forms a context, however this does not mean it must belong to exactly one object:

CSV/Excel file
Mood example

Mood example

- 1: Instances of **Song**
 - 2: Attribute **Name** English
 - 3: Attribute **run-time (seconds)**
 - 4: Attribute **Rating**
- 5: Relation **is contained by**
 - 6: Instances of **Album**
 - 7: Attribute **Name** English
 - 8: Attribute **Year**
- 9: Relation **has author**
 - 10: Instances of **Band**
 - 11: Attribute **Name** English
 - 12: Relation **has genre**
 - 13: Instances of **Music Genre**
 - 14: Attribute **Name** English

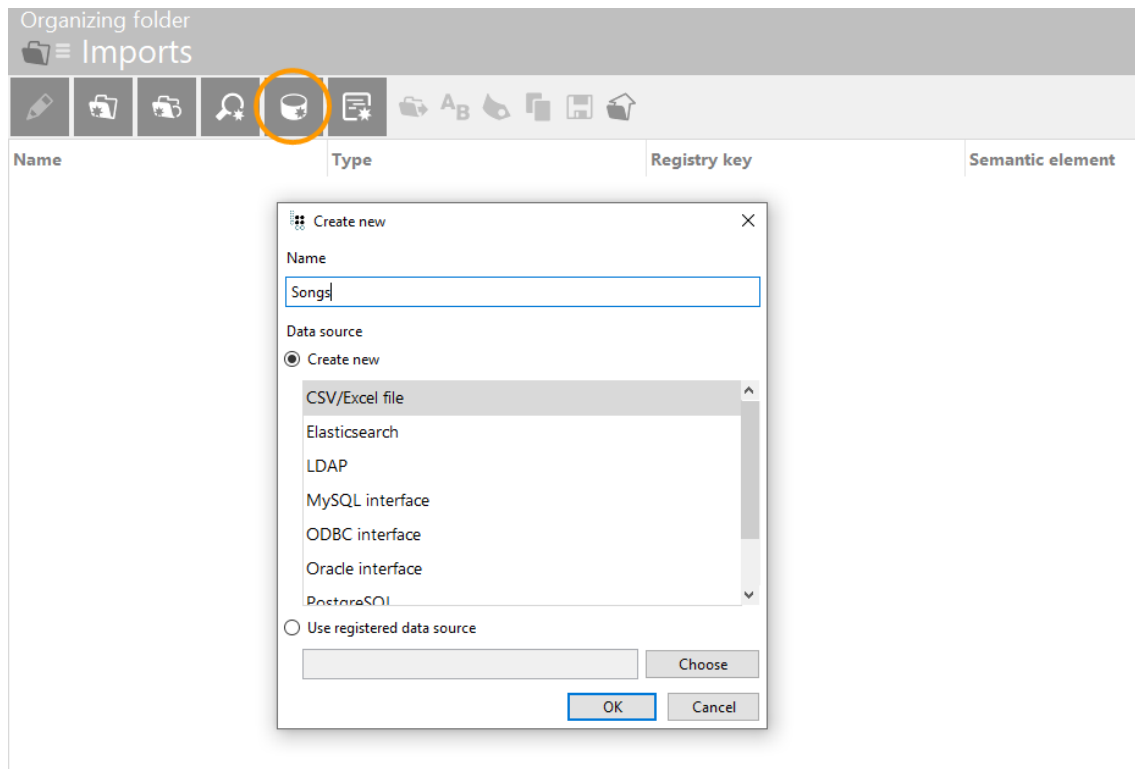


Everywhere that we build up new, specific objects and relation targets in our example, we must always specify at least one attribute for this object, in this case the respective name attribute that allows us to identify the corresponding object.



1.5.1.2 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.



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.



CSV/Excel file
Songs

Import file: C:\User\Desktop\Songs.csv ... Show table...

Export file: ... Show table...

Options

Table file type: CSV file

☒ 1st row contains heading ☒ Cell values are enclosed in quotes

Identify columns: ☒ by heading ☐ by position ☐ über Zeichenposition

Separator: ☐ Tab ☐ Space ☒ ;

Encoding: UTF-8

Line Separator: auto detect

Columns: Read from data source

Position	Heading	Field length	Type	Mappings	Identifier	Column
1	Title name	Variable	String			A
2	Artist	Variable	String			B
3	Album	Variable	String			C
4	Genre	Variable	String			D
5	Run-time	Variable	String			E
6	Year	Variable	String			F
7	My rating	Variable	String			G

☐ Edit Add column Remove columns Move up Move down Mappings

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.

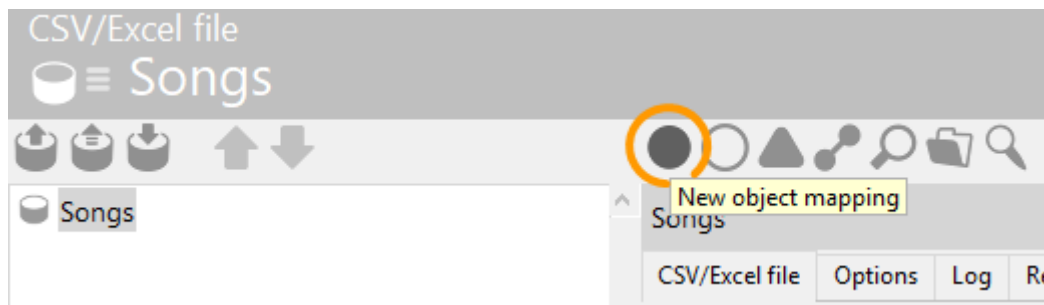
Separator: If a different separator than the default semicolon is used, this must also be specified when the column is not identified using the character position.

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 (»""«).

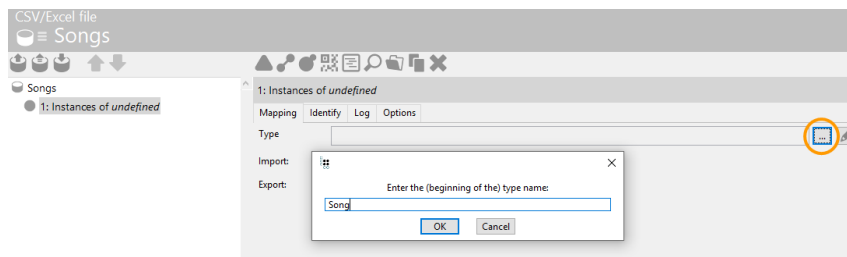
1.5.1.3 Definition of target structure and mappings

1.5.1.3.1 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.



The next step is to specify the type of object for import.



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.

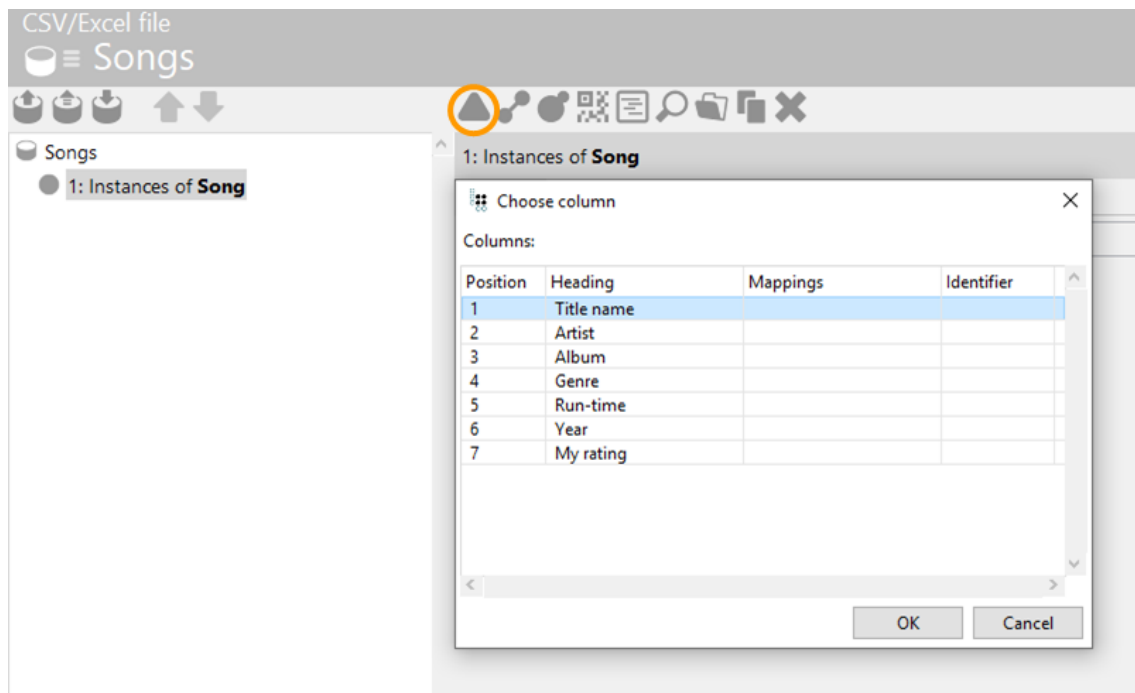
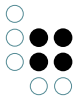
Exact type is specified by the following mapping: If the exact type to which the object is to be created is identified in the import source, this can be mapped here via the "New..." button. It must be a subtype of the type specified in the tab "Mapping".

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.

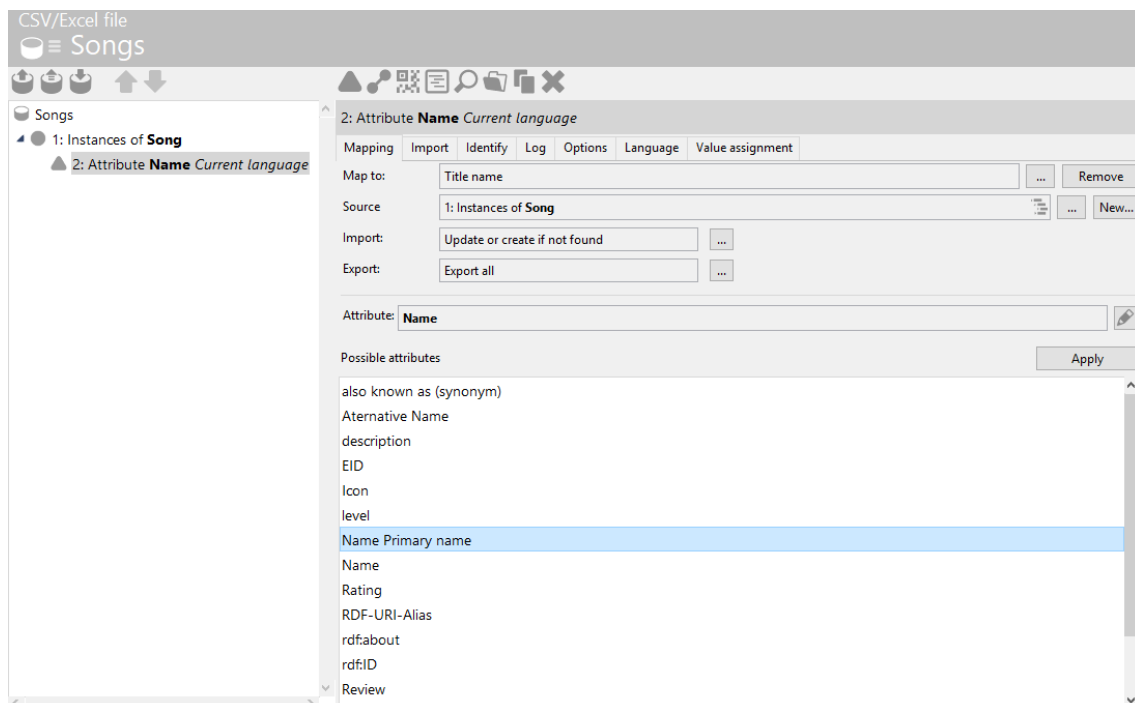
If you do not set the checkmark, the import will not be carried out for the multiple occurring objects and instead the user will be informed that the importer cannot uniquely identify the object.

1.5.1.3.2 The attribute mapping / Identifying objects

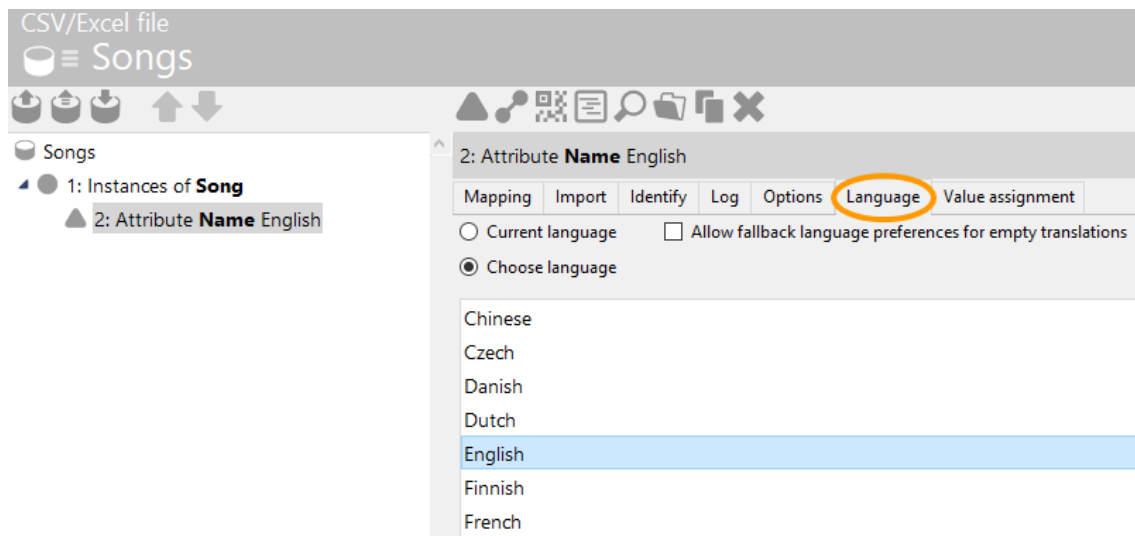
Now we want to link the information in the table to the object mapping of the songs. Attributes for individual songs are represented along with relations. In order to first create the track name for a song in the mapping, we add an attribute to the object mapping for song. Clicking on the "New attribute mapping" button opens a dialog, which must be used to select the relevant column from the table to be imported.



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.



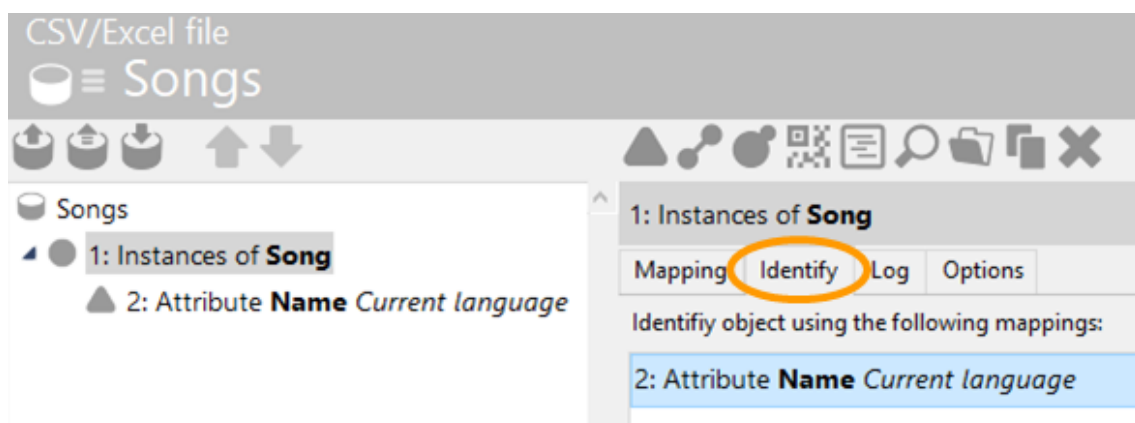
The first attribute created for an object is also automatically used for **identification of the object**. Note that for string attributes like the primary name, the language can be specified when translation layering is activated. When nothing else is specified, the current language (display language of the Knowledge-BUILDER) is automatically used as reference. The language can be specified within the language tab:



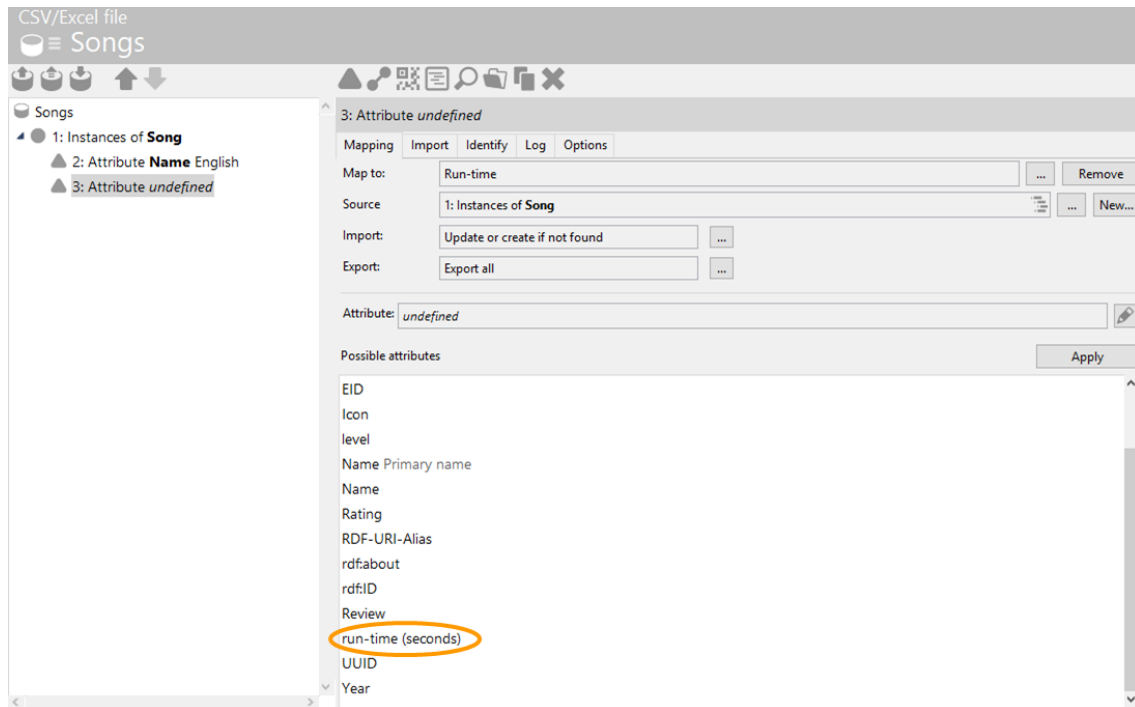
An object must be identified by at least one attribute - by its name or its ID, or by a combination of multiple attributes (as with the first and last name and date of birth of a person) - it should already exist so that it can be unambiguously found in the Knowledge Graph. This prevents unwanted duplicates 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.



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.



Mapping of translations

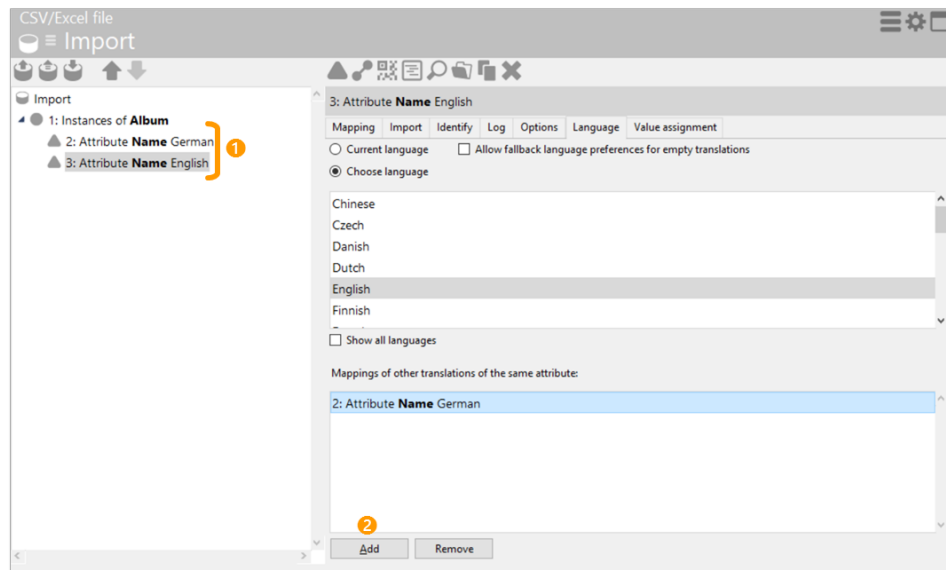
For string attributes with translations, e.g. the primary name of objects, we can define in which language the value needs to be imported.

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:



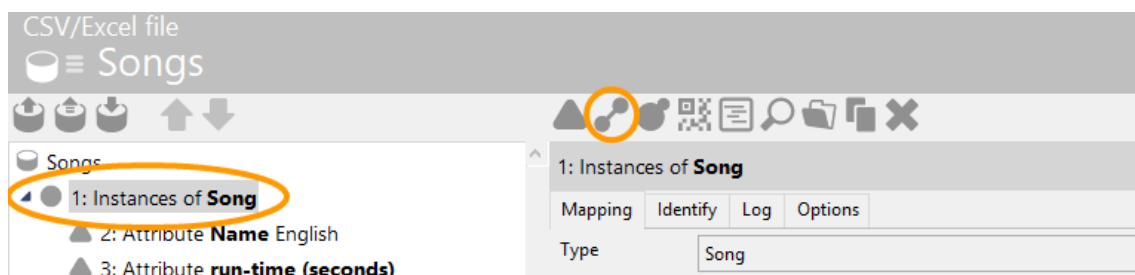
For each language, create a separate attribute mapping for the same attribute, but specify a different import language

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.

1.5.1.3.3 The relation mapping

Next, we want to map the album on which the song is located. Since albums are concrete 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."



Following that, just like for attributes, we get a list of all possible relations; and the required relation "is included in" is naturally included.



CSV/Excel file
Songs

Songs

- 1: Instances of **Song**
 - 2: Attribute **Name** English
 - 3: Attribute **run-time (seconds)**
 - 4: Relation **undefined**

4: Relation **undefined**

Mapping Import Export Identify Log Options

Source 1: Instances of **Song** New... Remove

Target New... Remove

Import: Update or create if not found

Export: Export all

Relation undefined

Possible relations Apply

- has remixed version
- has style
- is contained by**
- is cover version of
- is marked up in
- is object of
- is remix version of

Inverse relations undefined

Possible inverse relations Apply

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.

CSV/Excel file
Songs

Songs

- 1: Instances of **Song**
 - 2: Attribute **Name** English
 - 3: Attribute **run-time (seconds)**
 - 4: Relation **is contained by**

4: Relation **is contained by**

Mapping Import Export Identify Log Options

Source 1: Instances of **Song** New... Remove

Target New... Remove

Import: Update or create if not found

Export: Export all

Relation **is contained by**

Possible relations Apply

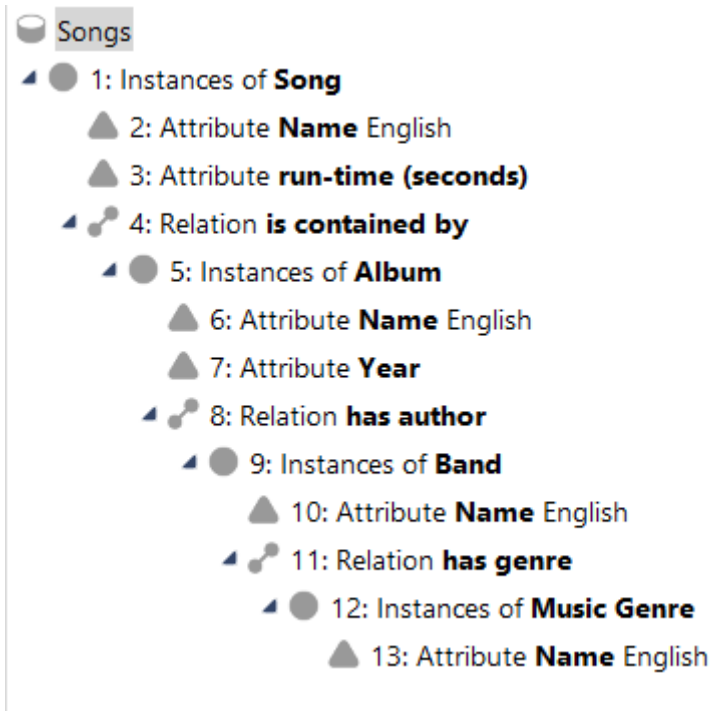
- has remixed version
- has style
- is contained by**
- is cover version of
- is marked up in
- is object of
- is remix version of

Inverse relations **contains**

Possible inverse relations Apply

contains

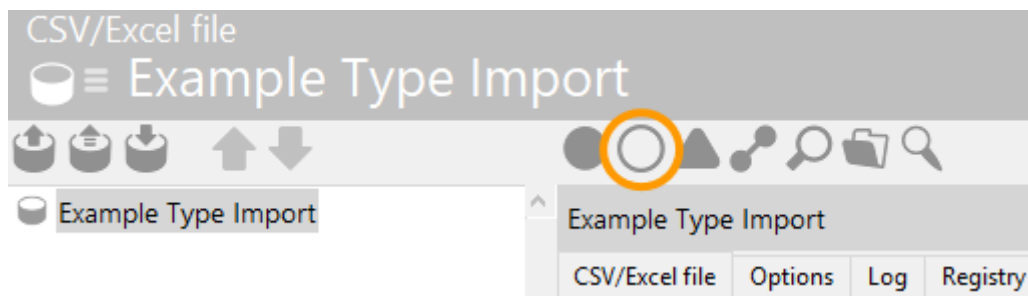
For new object mappings, we then once again have to select the attribute that identifies the target object etc. This creates the target structure of the import.



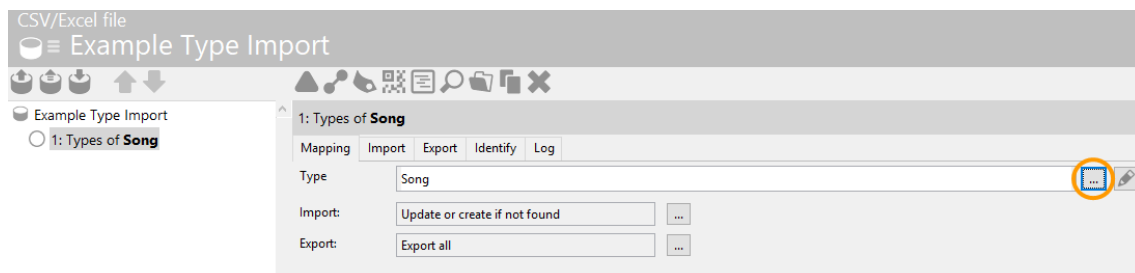
1.5.1.3.4 The type mapping

Types can also be imported and exported. Let's assume we want to import the genres of songs as types.

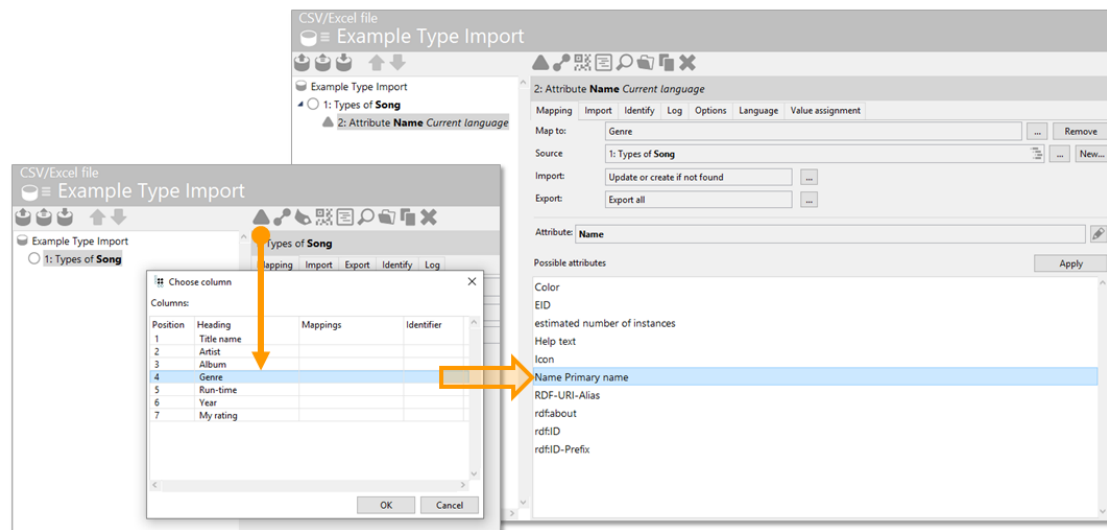
To map a new type, we choose the "New type mapping" button.



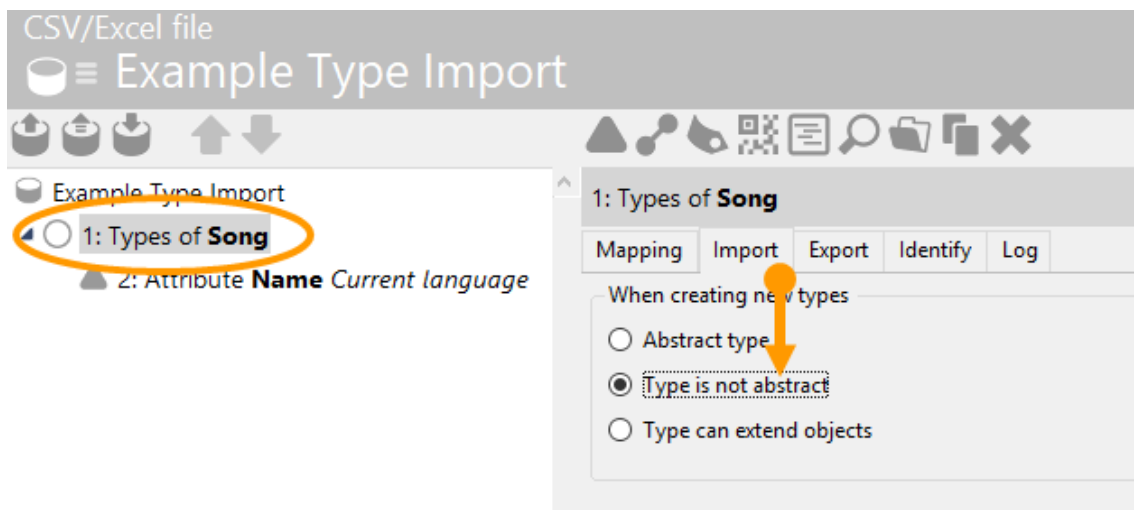
Following this, we have to specify the super-type of the new types to be created, in our example, the super-type would be "Song:"



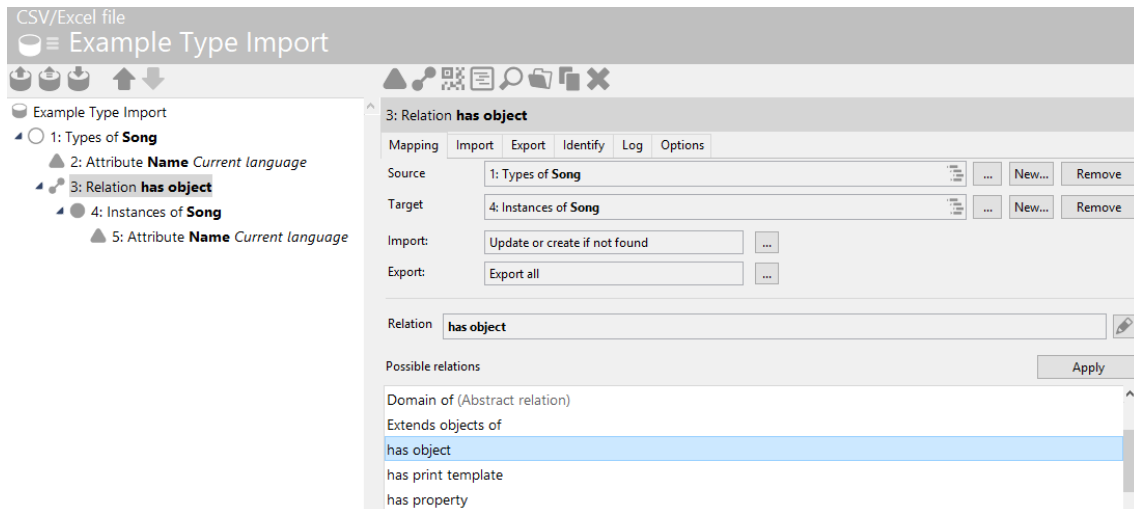
Following that, we have to specify from which column of the imported table the name of our new types is to be taken:



Following that, we still have to specify on the “Import” tab that our new types are not supposed to be abstract:



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.

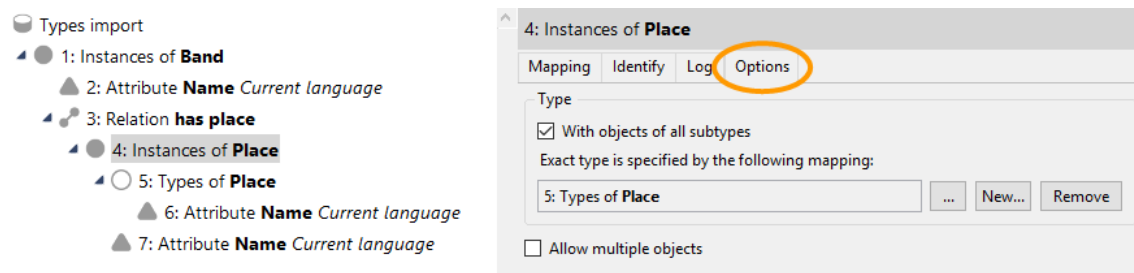


If we now import this mapping, we get the desired result. The songs that already exist in the Knowledge Graph are taken into account by the import setting “Update or create if not found” and moved under their respective type so that no object is created twice (see chapter Import behavior settings). A 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.

Person/Band	Origin	Type of location
Paul McCartney	Liverpool	City
The Beatles	Great Britain	Country

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.



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 knowledge 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.

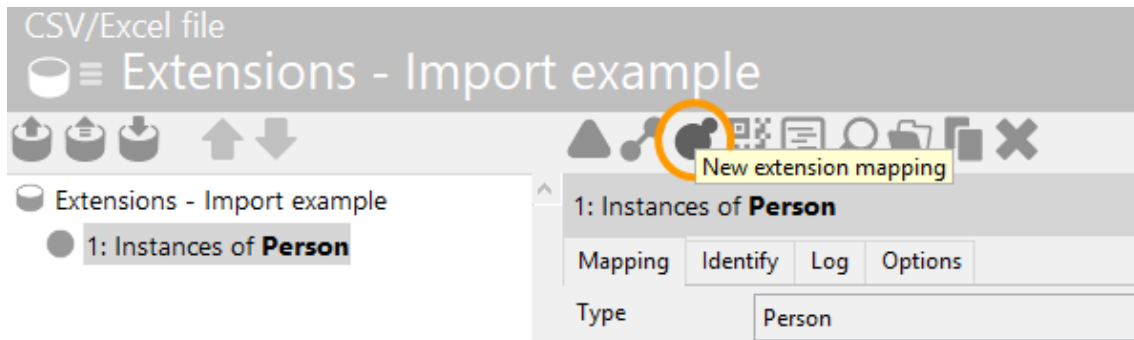
1.5.1.3.5 Mapping of extensions

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:

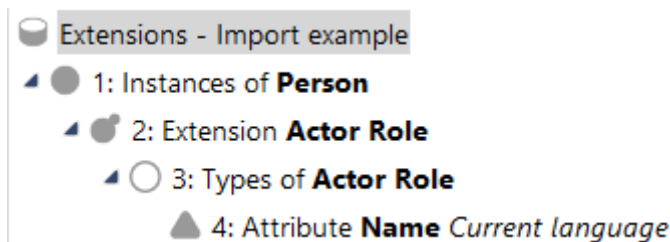


Person	Band	Rolle
Ron Wood	Faces	Guitarist
Ron Wood	Jeff Beck Group	Bassist
Ron Wood	Rolling Stones	Guitarist

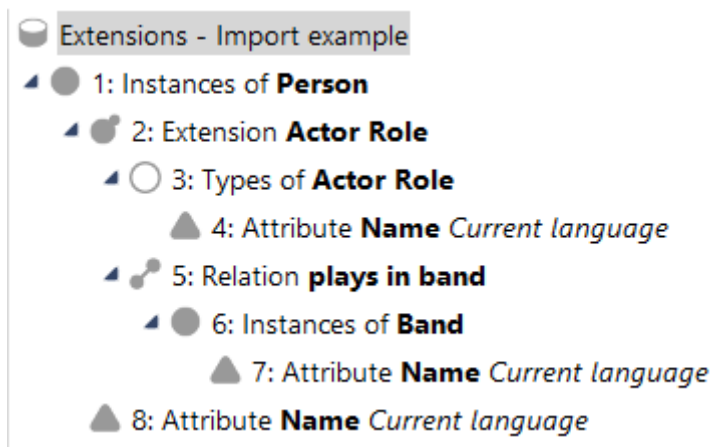
Ron Wood is a guitarist with the Faces and the Rolling Stones, but a bassist with the Jeff Beck Group. In order to map this, we must select the object for which an extension was defined in the schema and then press the “New extension mapping” button.



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).



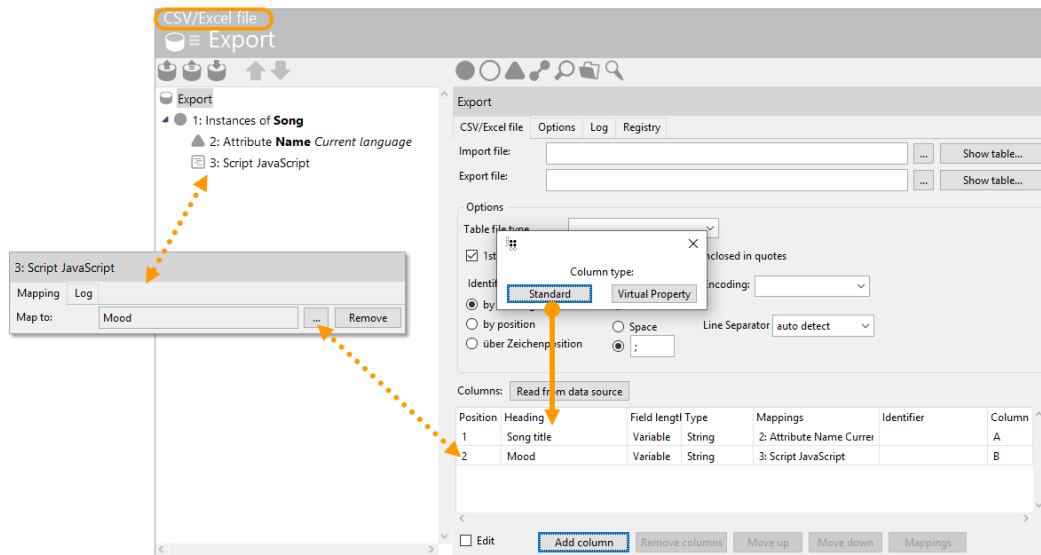
As with objects and types, the relation can be mapped to the extension (or to the subtypes of an extension).



1.5.1.3.6 The script mapping

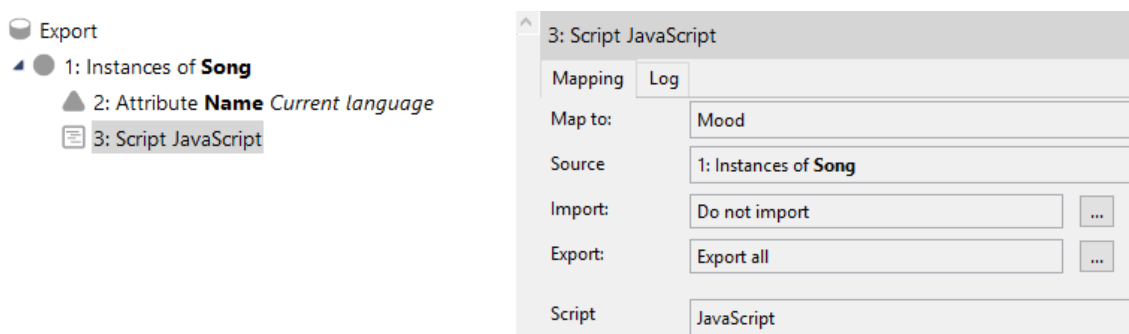
Note: The script mapping can only be used upon export. The script can be written in either JavaScript or KScript. Export mappings are only available in forms of a CSV/Excel mapping.

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 assign the output column ("Map to"):



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:



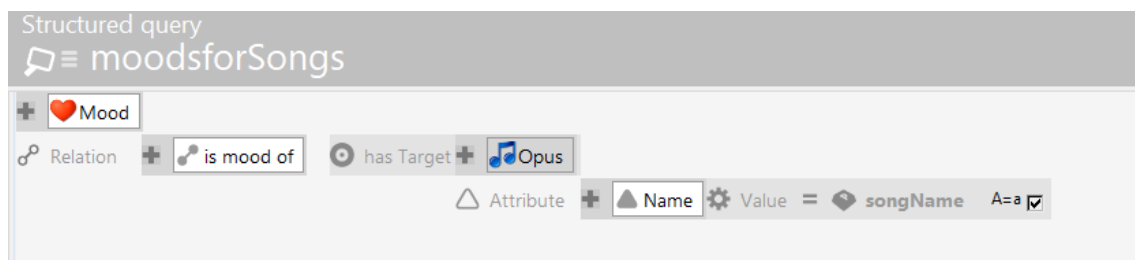
To generate the second column, we require the following script:

```
function exportValueOf(element) {
    var mood = "";
    var relTargets = $k.Registry.query("moodsforSongs").findElements({songName: element.attributeV
    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"):



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:

Song title	Mood
Black Country Rock	
19 th Nervous Breakdown	
A Maniac Depressive Named Laughing Boy	
A Place For My Head	aggressive
All the Madmen	
Bipolar	
Bleed It Out	
Bleed Like Me	
Breaking The Habit	
By Myself	aggressive
Back To Black	dramatic, bittersweet, swinging
China Girl (Bowie)	melancholic/dull, cold
Climbing up the Walls	
Crawling	aggressive
Creep	anthemic, elegiac, dramatic, lethargic, melancholic/dull
Digging In The Dirt	

1.5.1.4 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).

	A	B	C	D	E
1	Title name	Genre	Mood	Run-time	Year
2	Eleanor Rigby	Oldies	reflective, dreamy	127	1966
3	For No One	Oldies	acerbic	121	1966
4	I'm Only Sleeping	Oldies	quirky, mellow	181	1966
5	Yellow Submarine	Oldies	spacey, trippy, playful	160	1966

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".

CSV/Excel file

Songs

1: Instances of **Song**

- 2: Attribute **Name** *Current language*
- 3: Attribute **run-time (seconds)**
- 4: Relation **has mood**
- 5: Instances of **Mood**
- 6: Attribute **Name** *Current language*

Songs

CSV/Excel file Options Log Registry

Import

- ☐ Import in a single transaction
- ☒ Use multiple transactions for import ☒ Update metrics
- ☒ Triggers activated
- ☐ Automatic generation of name for nameless objects

Data source

- ☐ Read full table (contains forward references)
- ☒ Read row by row (no forward references)

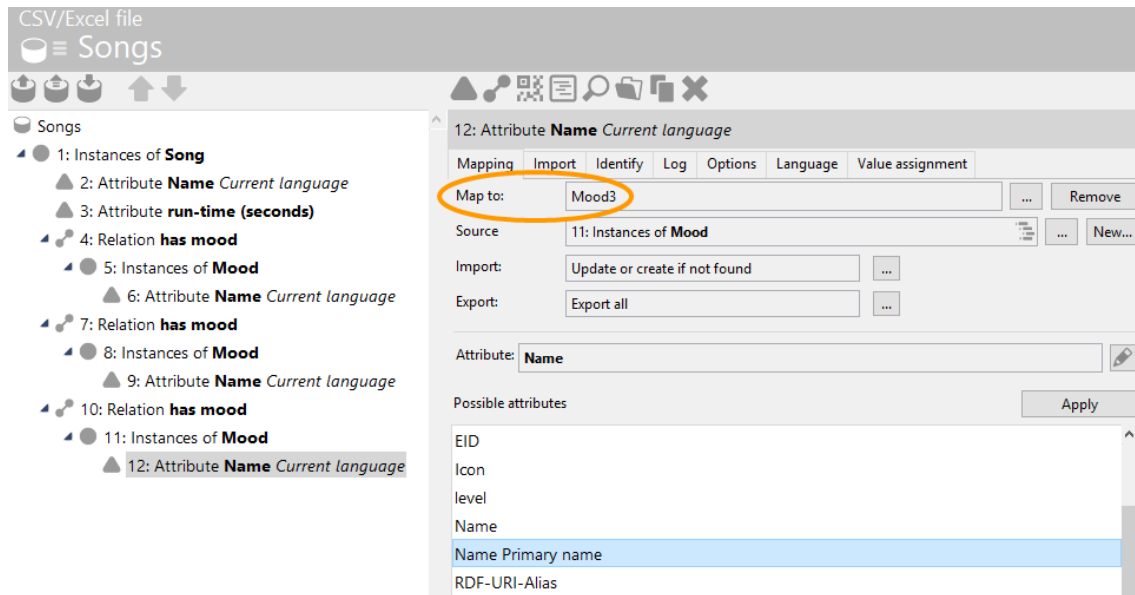
Separator within one cell:

Column	Separator
Title name	
Genre	
Mood	,
Run-time	

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.

	A	B	C	D	E	F	G
1	Title name	Genre	Mood	Mood2	Mood3	Run-time	Year
2	Eleanor Rigby	Oldies	reflective	dreamy		127	1966
3	For No One	Oldies	acerbic			121	1966
4	I'm Only Sleeping	Oldies	quirky	mellow		181	1966
5	Yellow Submarine	Oldies	spacey	trippy	playful	160	1966

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".



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.

	A	B	C	D	E
1	Title name	Genre	Mood	Run-time	Year
2	Eleanor Rigby	Oldies	reflective	127	1966
3	Eleanor Rigby		dreamy	127	1966
4	For No One	Oldies	acerbic	121	1966
5	I'm Only Sleeping	Oldies	quirky	181	1966
6	I'm Only Sleeping		mellow	181	1966
7	Yellow Submarine	Oldies	spacey	160	1966
8	Yellow Submarine		trippy	160	1966
9	Yellow Submarine		playful	160	1966

In this case, no special import settings are required, as the system identifies the object using the identifying attribute and creates the relations correctly.

1.5.1.5 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.

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

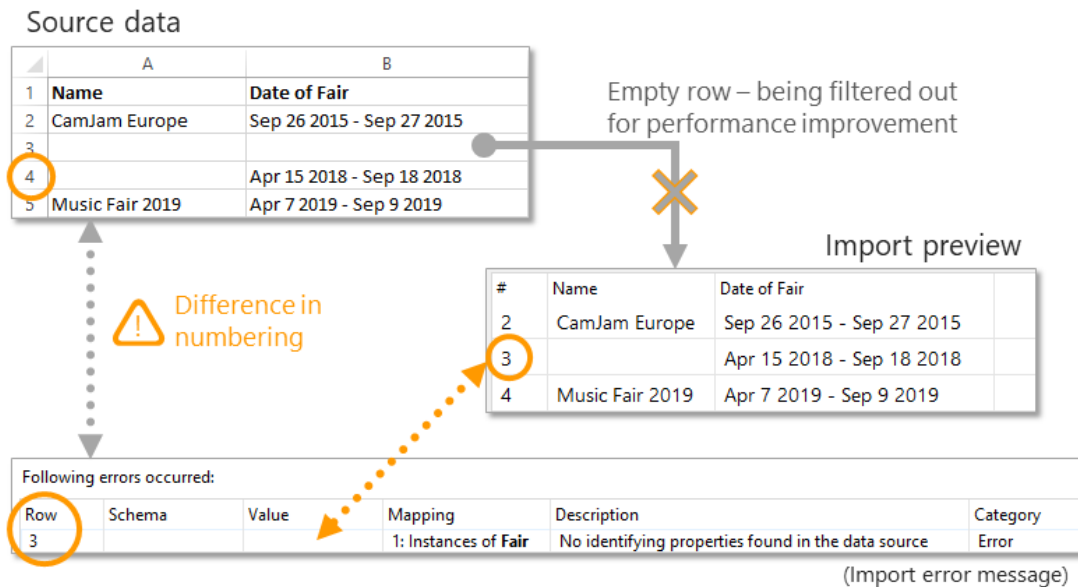
The import mapping of the i-views Knowledge-BUILDER is a row-based system.

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. Since the empty rows don’t carry any information, they are not being analyzed (avoiding unnecessary processing load for comparison with the Knowledge Graph). 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).



1.5.1.5.1 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.



About A Girl

Song

Attributes

Name

About A Girl

Rating

6

Add attribute

Relations

has genre

Alternative Rock

Add relation

The song "About A Girl" is already available in the Knowledge Builder.

Song	Run-time	Rating	Author
About A Girl	168	5	Nirvana

The import table contains information on the length, rating and creator of the song.

CSV/Excel file

Update

Update

1: Instances of **Song**

2: Attribute **Name** *Current language*

3: Attribute **run-time (seconds)**

4: Attribute **Rating**

5: Relation **has author**

6: Instances of **Band**

7: Attribute **Name** *Current language*

1: Instances of **Song**

Mapping

Identify

Log

Options

Type

Song

Import:

Update

...

Export:

Export all

...

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."



About A Girl

Song

Attributes

run-time (seconds)

≡

168

▶ Name

≡

About A Girl

Rating

≡

5

Add attribute

Relations

has genre

≡

Alternative Rock

has author

≡

Nirvana

Add relation

The result: The song has been updated and has received new attributes and relations. Already existing properties have been updated (value).

1.5.1.5.2 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.

1.5.1.5.3 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.



About A Girl

Song

Attributes

run-time (seconds)

168

▶ Name

About A Girl

Rating

5

Add attribute

Relations

has author

Nirvana

has author

Nirvana

Add relation

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.

1.5.1.5.4 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.

Song	Run-time
19 th Nervous Breakdown	113
A Maniac Depressive Named Laughing Boy	300
A Place For My Head	249
About A Girl	168

For most songs, the duration in the import table differs...

Name	[3] run-time (seconds)
19th Nervous Breakdown	113
A Manic Depressive Named Lauging Boy	300
A Place for my Head	249
About A Girl	168



... from the duration of the songs in the database.

Delete values of same type

- 1: Instances of **Song**
- 2: Attribute **Name** *Current language*
- 3: Attribute **run-time (seconds)**

3: Attribute **run-time (seconds)**

Mapping

Identify

Log

Options

Language

Value assignment

Map to: Run-time

Source: 1: Instances of **Song**

Import: Delete all of same kind

Export: Export all

Remove

New...

For the attribute "Duration" we use the import setting "Delete all of the same type."

Name	run-time (seconds)
19th Nervous Breakdown	.
A Manic Depressive Named Laughing Boy	.
A Place for my Head	.
About A Girl	.

After the import, all attribute values of the attribute type duration have been deleted for these 4 songs.

1.5.1.5.5 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.

1.5.1.5.6 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.

Following errors occurred:

Row	Schema	Value	Mapping	Description	Category
2	Run-time	120	3: Attribute run-time (seconds)	Attribute "run-time (seconds)" cannot be added to '19th Nervous Breakdown'	Error
3	Run-time	306	3: Attribute run-time (seconds)	Attribute "run-time (seconds)" cannot be added to 'A Maniac Depressive Named Laughing Boy'	Error
4	Run-time	239	3: Attribute run-time (seconds)	Attribute "run-time (seconds)" cannot be added to 'A Place For My Head'	Error
5	Run-time	168	3: Attribute run-time (seconds)	Attribute "run-time (seconds)" cannot be added to 'About A Girl'	Error

1.5.1.5.7 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.

1.5.1.5.8 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 with 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.

1.5.1.5.9 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.

1.5.1.5.10 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.

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):

Name	Date of Fair		A	B
CamJam Europe	Sep 26 2015 - Sep 27 2015	1	Name	Date of Fair
choir.com Fair	Apr 1 2015 - Apr 4 2015	2	CamJam Europe	Sep 26 2015 - Sep 27 2015
Music Fair 2018	Apr 15 2018 - Sep 18 2018	3		
Music Fair 2019	Apr 7 2019 - Sep 9 2019	4		Apr 15 2018 - Sep 18 2018
		5	Music Fair 2019	Apr 7 2019 - Sep 9 2019

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:



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:

Following errors occurred:

Row	Schema	Value	Mapping	Description	Category
3			1: Instances of Fair	No identifying properties found in the data source	Error

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:



Name	Date of Fair
CamJam Europe	Sep 26 2015 - Sep 27 2015
Music Fair 2019	Apr 7 2019 - Sep 9 2019

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.

1.5.1.6 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 to 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*.

Expression	Description	Example	Input	Output
<np>	Print output of content of column n	Hits: <1p>	1 (integer) none (string)	Hits: 1 Hits: none



<ns>	Output of string in column n	Hello <1s>!	'Peter'	Hello Peter!
<nu>	Output of string in column n in upper case	Hello <1u>!	'Peter'	Hello PETER!
<nl>	Output of string in column n in lower case	Hello <1l>!	'Peter'	Hello peter!
<ncstart-stop>	Partial string from position start to stop from column n	<1c3-6> <1c3> <1c3->	Columns	olum umn lums
<nmregex>	Test whether the content of column n matches the regex regular expression. The following expressions are only evaluated if the regular expression applies.	<1m0[0-9]>hi <1m\$>test	01 123 (blank) 123	hi (blank) test (blank)
<nxregex>	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.	<1x0[0-9]>hello	01 123	(blank) hello
<neregex>	Selects all hits for regex from the contents of column n. Individual hits are separated by commas in the result.	<1eL+> <1e\d\d\d>	HELLO WORLD 02.10.2001	LL,L 2001
<nrregex>	Removes all hits for regex from the contents of column n	<1rL>	HELLO WORLD	HEO WORD
<ngregex>	Transmits the contents of all groups of the regular expression	<1g\+(\d+)\->	+42-13	42
<nfformat>	Formats numbers, date and time specifications from column n according to the format format specification	<1f#,0.00> <1fd/m/y> <1fdd/mmm>	3.1412 1234.5 1 May 1935 1 May 1935	3.14 1234.50 1/5/1935 01/May



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).

Ex- pre- sion	Description	Ex- an- ple	Ir- re- sult	Out- put
<\$name>	Refers to a column by means of a unique column identifier <i>name</i> and subsequent transformation by means of the expression. The \$ characters are a functional component of the identifier syntax.	<\$name>	name	name

For more information on how to use regular expressions (regEx), see <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:

#	Media	Item number	Title	Artist	Genre	<1mCD><2c1-3>000	<1xCD><1xMD><2c1-4>00	Playlist Summer 2019
2	CD	010000	The suburbs	Arcade Fire	Postwave	010000		Playlist Summer 2019
3	CD	010100	Modern Man	Arcade Fire	Postwave	010000		Playlist Summer 2019
4	LP	010101	Empty Room	Arcade Fire	Postwave		010100	Playlist Summer 2019
5	MP3	010102	Half Light I	Arcade Fire	Postwave		010100	Playlist Summer 2019
6	OGG	010103	Half Light II	Arcade Fire	Postwave		010100	Playlist Summer 2019
7	LP	010104	Month Of Day	Arcade Fire	Postwave		010100	Playlist Summer 2019
8	LP	010105	Deep Blue	Arcade Fire	Postwave		010100	Playlist Summer 2019
9	LP	010106	Eleanor Rigby	The Beatles	Oldies		010100	Playlist Summer 2019
10	LP	010107	I Want To Tell You	The Beatles	Oldies		010100	Playlist Summer 2019
11	LP	010109	I'm Only Sleeping	The Beatles	Oldies		010100	Playlist Summer 2019
12	LP	010110	Love To You	The Beatles	Oldies		010100	Playlist Summer 2019
13	MD	010200	Taxman	The Beatles	Oldies			Playlist Summer 2019
14	LP	010201	About A Girl	Nirvana	Rock		010200	Playlist Summer 2019

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:

Enable the editing of columns first

Add the identifier name for the column (the identifier "media" always will stick to the column with the title "Media")

Click on the "Add column" button

Choose the virtual property

For the heading, enter the column identifier in combination with the regular expression

To ensure that the current data is loaded, click on "Read from data source"

Click on "Show table" to see the result

The screenshot shows the 'Playlist' application window. The 'Options' dialog is open, showing 'Table file type' set to 'Excel file (xlsx)' and '1st row contains heading' checked. Under 'Identify columns', 'by heading' is selected. A 'Column type' dialog is also open, showing 'Virtual Property' selected. The 'Columns' table at the bottom lists 8 columns. Column 6 has a heading '<1u>' and type 'virtuell'. Column 7 has a heading '<\$media\$(mp3|ogg)*.<\$media\$>' and type 'virtuell'. The 'Read from data source' button is highlighted with a red circle 6. The 'Add column' button is highlighted with a red circle 3. The 'Edit' checkbox is checked, and the 'Mappings' button is visible.

Position	Heading	Field length	Type	Mappings	Identifier	Column
1	Media	Variable	String		media 2	A
2	Item number	Variable	String			B
3	Title	Variable	String			C
4	Artist	Variable	String			D
5	Genre	Variable	String			E
6	<1u>	Variable	virtuell			F
7	<\$media\$(mp3 ogg)*.<\$media\$> 5	Variable	virtuell			G
8	Playlist Summer 2019	Variable	virtuell			H

A click on the "Show table" button shows the preview with the transformed column entries:



#	Media	Item number	Title	Artist	Genre	<1u>	<\$media\$m(mp3 ogg)*.<\$media\$I>	Playlist Summer 2019
2	CD	010000	The suburbs	Arcade Fire	Postwave	CD		Playlist Summer 2019
3	CD	010100	Modern Man	Arcade Fire	Postwave	CD		Playlist Summer 2019
4	LP	010101	Empty Room	Arcade Fire	Postwave	LP		Playlist Summer 2019
5	mp3	010102	Half Light I	Arcade Fire	Postwave	MP3	*.mp3	Playlist Summer 2019
6	ogg	010103	Half Light II	Arcade Fire	Postwave	OGG	*.ogg	Playlist Summer 2019
7	LP	010104	Month Of Day	Arcade Fire	Postwave	LP		Playlist Summer 2019
8	LP	010105	Deep Blue	Arcade Fire	Postwave	LP		Playlist Summer 2019
9	LP	010106	Eleanor Rigby	The Beatles	Oldies	LP		Playlist Summer 2019
10	LP	010107	I Want To Tell You	The Beatles	Oldies	LP		Playlist Summer 2019
11	LP	010109	I'm Only Sleeping	The Beatles	Oldies	LP		Playlist Summer 2019
12	LP	010110	Love To You	The Beatles	Oldies	LP		Playlist Summer 2019
13	MD	010200	Taxman	The Beatles	Oldies	MD		Playlist Summer 2019
14	LP	010201	About A Girl	Nirvana	Rock	LP		Playlist Summer 2019

The following figure shows the effect of swapped columns in an import table: If only column numbers are used like in <1u>, the wrong column is accidentally 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:

#	Item number	Media	Genre	Title	Artist	<1u>	<\$media\$m(mp3 ogg)*.<\$media\$I>	Playlist Summer 2019
2	010000	CD	Postwave	The suburbs	Arcade Fire	010000		Playlist Summer 2019
3	010100	CD	Postwave	Modern Man	Arcade Fire	010100		Playlist Summer 2019
4	010101	LP	Postwave	Empty Room	Arcade Fire	010101		Playlist Summer 2019
5	010102	mp3	Postwave	Half Light I	Arcade Fire	010102	*.mp3	Playlist Summer 2019
6	010103	ogg	Postwave	Half Light II	Arcade Fire	010103	*.ogg	Playlist Summer 2019
7	010104	LP	Postwave	Month Of Day	Arcade Fire	010104		Playlist Summer 2019
8	010105	LP	Postwave	Deep Blue	Arcade Fire	010105		Playlist Summer 2019
9	010106	LP	Oldies	Eleanor Rigby	The Beatles	010106		Playlist Summer 2019
10	010107	LP	Oldies	I Want To Tell You	The Beatles	010107		Playlist Summer 2019
11	010109	LP	Oldies	I'm Only Sleeping	The Beatles	010109		Playlist Summer 2019
12	010110	LP	Oldies	Love To You	The Beatles	010110		Playlist Summer 2019
13	010200	MD	Oldies	Taxman	The Beatles	010200		Playlist Summer 2019
14	010201	LP	Rock	About A Girl	Nirvana	010201		Playlist Summer 2019

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\$I> 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\$I> afterwards.

The complete regular expression <\$media\$m(mp3|ogg)*.<\$media\$I> returns the intended result, whereas another sequence of the expressions *.<\$media\$I><\$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:



#	Media	Item number	Title	Artist	Genre	<1u>	*.<\$media\$!><\$media\$m(mp3 ogg)>	Playlist Summer 2019
2	CD	010000	The suburbs	Arcade Fire	Postwave	CD	*.cd	Playlist Summer 2019
3	CD	010100	Modern Man	Arcade Fire	Postwave	CD	*.cd	Playlist Summer 2019
4	LP	010101	Empty Room	Arcade Fire	Postwave	LP	*.lp	Playlist Summer 2019
5	mp3	010102	Half Light I	Arcade Fire	Postwave	MP3	*.mp3	Playlist Summer 2019
6	ogg	010103	Half Light II	Arcade Fire	Postwave	OGG	*.ogg	Playlist Summer 2019
7	LP	010104	Month Of Day	Arcade Fire	Postwave	LP	*.lp	Playlist Summer 2019
8	LP	010105	Deep Blue	Arcade Fire	Postwave	LP	*.lp	Playlist Summer 2019
9	LP	010106	Eleanor Rigby	The Beatles	Oldies	LP	*.lp	Playlist Summer 2019
10	LP	010107	I Want To Tell You	The Beatles	Oldies	LP	*.lp	Playlist Summer 2019
11	LP	010109	I'm Only Sleeping	The Beatles	Oldies	LP	*.lp	Playlist Summer 2019
12	LP	010110	Love To You	The Beatles	Oldies	LP	*.lp	Playlist Summer 2019
13	MD	010200	Taxman	The Beatles	Oldies	MD	*.md	Playlist Summer 2019
14	LP	010201	About A Girl	Nirvana	Rock	LP	*.lp	Playlist Summer 2019

1.5.1.7 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:

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 "Or-



acle 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 Example of the CSV file.

1.5.1.8 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:

```
<?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.

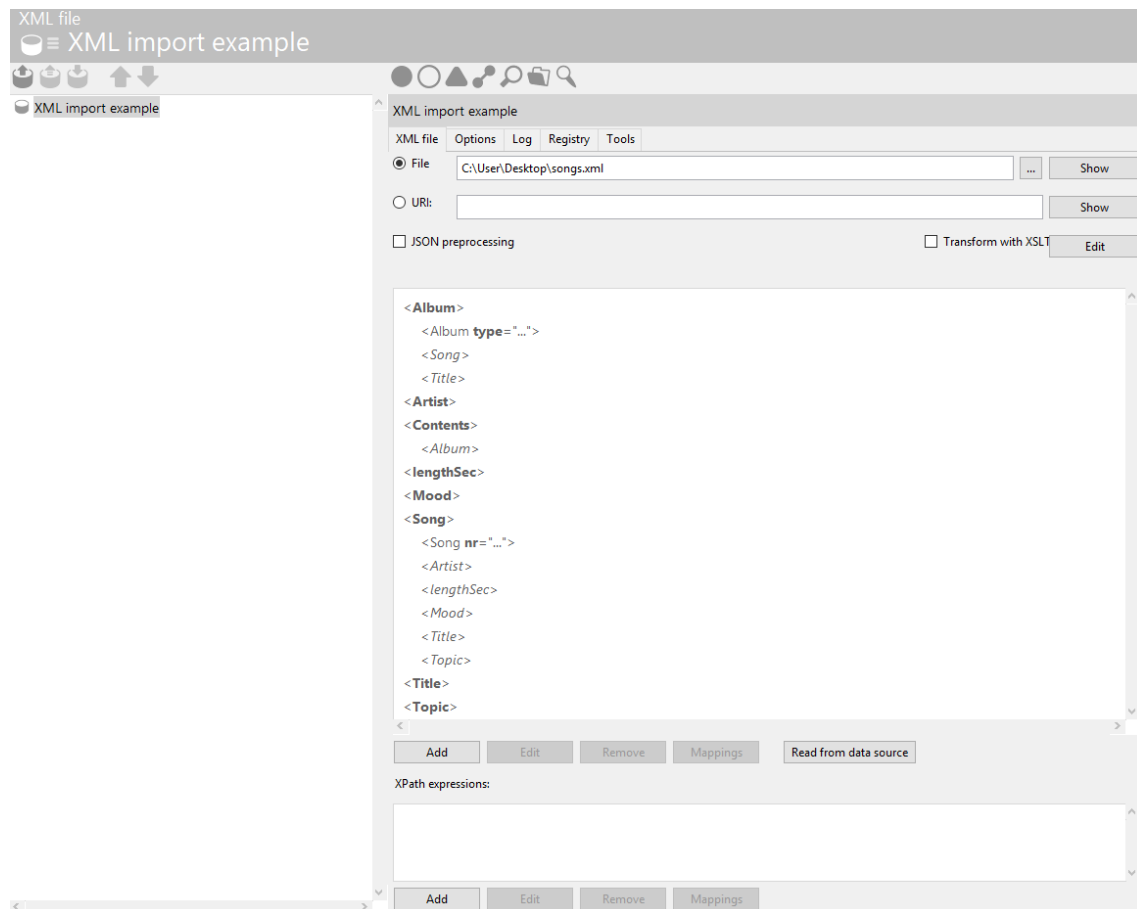
JSON preprocessing makes it possible to convert a JSON file to XML before the actual import.

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.

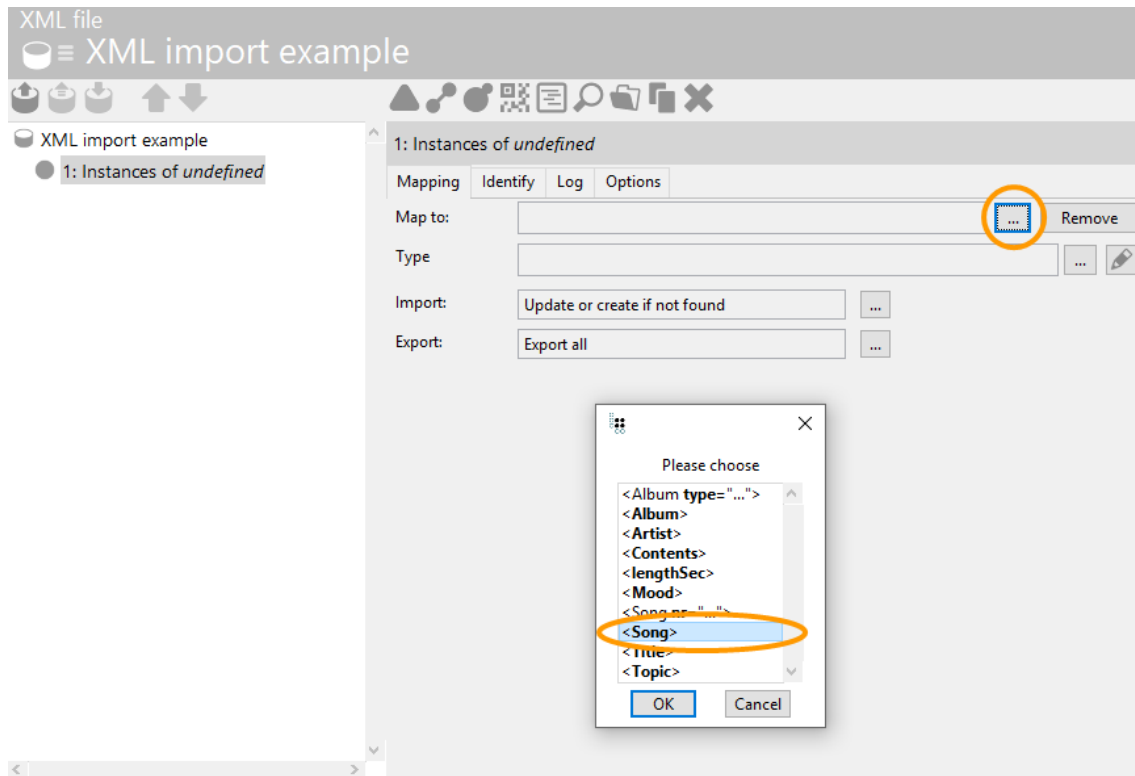
Once the file has been selected, use the "Read from data source" button to read out the XML



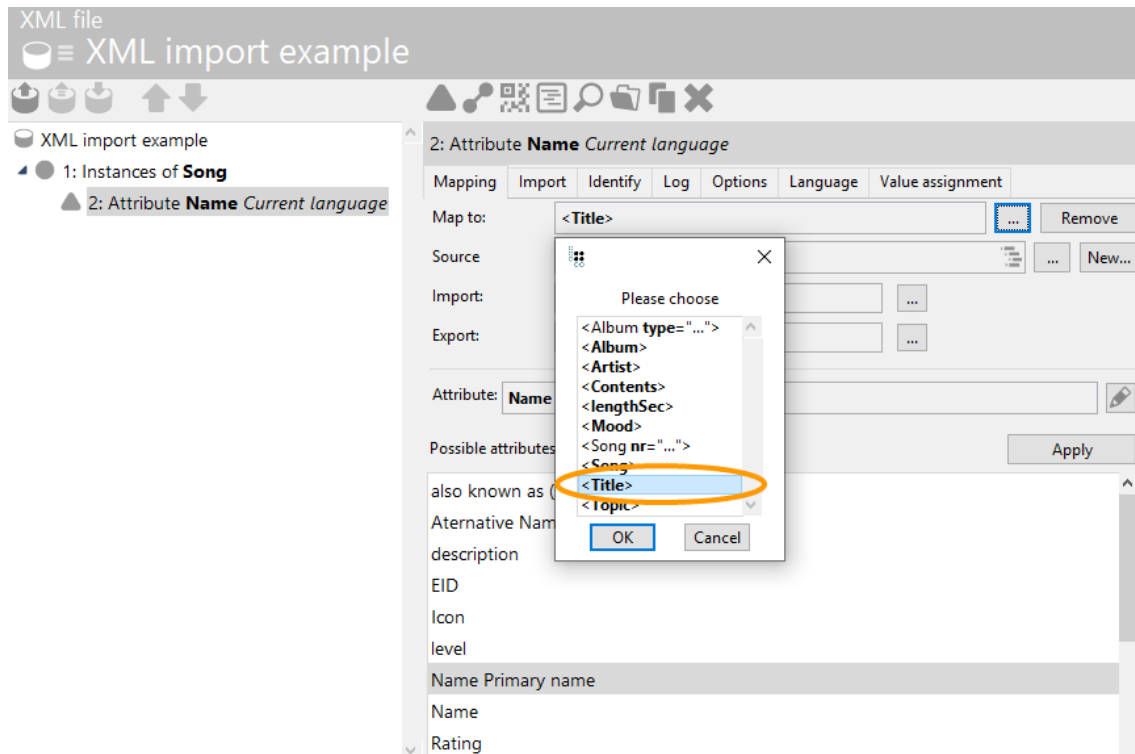
structure, which is then displayed in the right-hand window.



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.



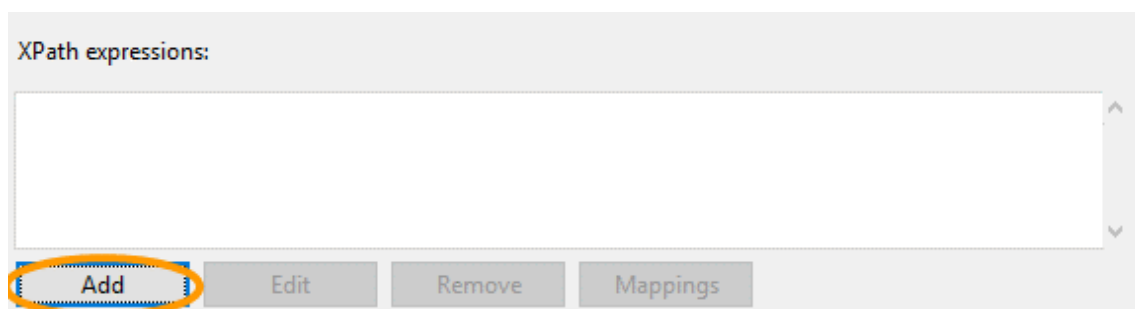
As our example shows, the tags are not always unambiguous without context: `<Title>` is used for both album titles and song titles. The object type only becomes clear in combination with the surrounding tag. Often the context of the XML structure and the context of the mapping hierarchy are synchronous: 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.



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.



The correct XPath expression is:

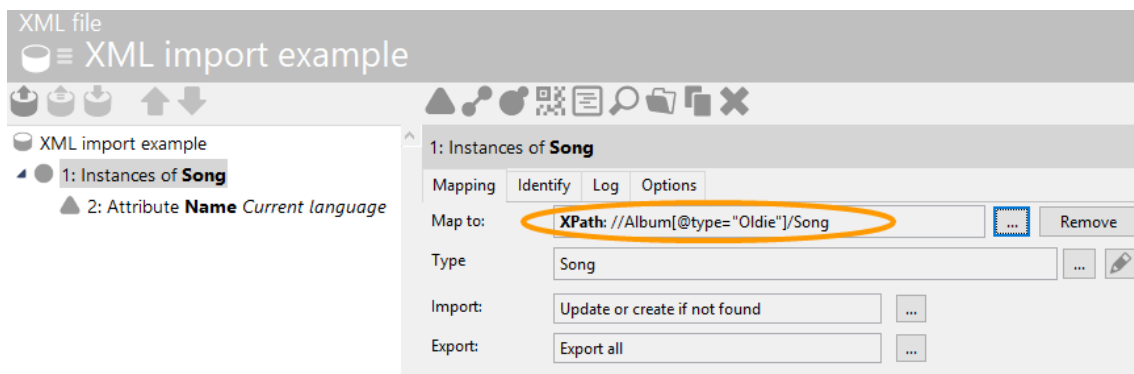
```
//Album[@type="Oldie"]/Song
```

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.



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.

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".
	ances- tor::termEntry/attribute::id	Superordinate node on any level with name ("termEntry") and attribute ("id")
	/myparent/mychild[text()]	Text between certain tags

1.5.1.9 Further options, log and registry

1.5.1.9.1 Further options at the import

In the "Options" tab, the following functions are available for selection irrespective of the data source:

Import
☐ Import in a single transaction
☒ Use multiple transactions for import ☒ Update metrics
☒ Triggers activated
☐ Automatic generation of name for nameless objects



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:

Data source

☐ Read full table (contains forward references)

☒ Read row by row (no forward references)

Separator within one cell:

Column	Separator
Media	
Item number	
Title	
Artist	

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 avail-



able, 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:

Data source

☒ Incremental XML import

Partitioning element: ...

☐ File in DTD

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).

1.5.1.9.2 Log

The functions in the “Log” tab allow changes that are made upon import to be tracked.

CSV/Excel file Options Log Registry

☐ Add created semantic elements to a folder

☐ Add modified semantic elements to a folder

☐ Add affected semantic elements to a folder

☐ New folder

☒ Folder ...

☐ Write errors to a file

Letzer Import

Letzer Export

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.

1: Instances of **Song**

Mapping Identify Log Options

☒ Dont categorize log entries

☐ Category of log entries

☐ Write value in error logs

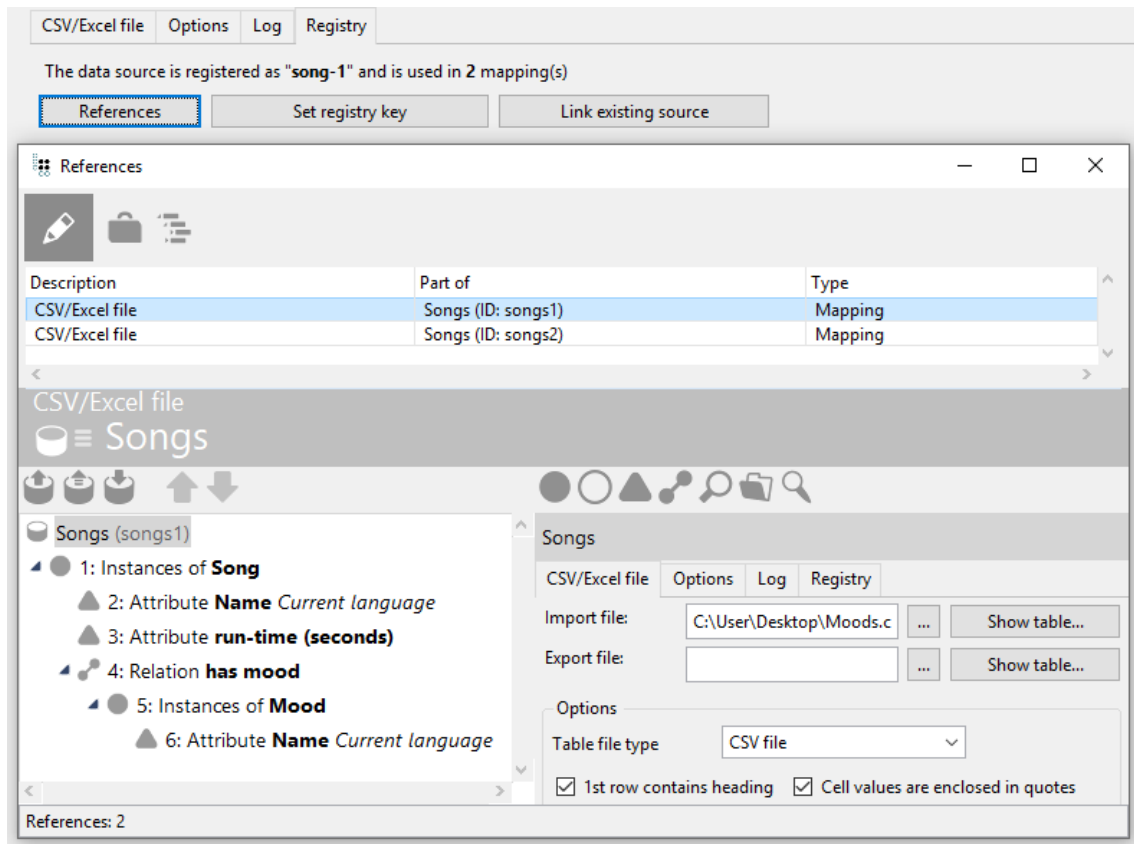
The “Log” tab is also available in the case of the individual mapping objects. When necessary, a category can be entered for log entries here. Moreover, it is possible to define that the value of the corresponding object/corresponding property should be written into the error log. This is not activated by default, in order to avoid revealing sensitive data (e.g. passwords).

1.5.1.9.3 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:



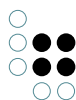
1.5.2 Attribute types and formats

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.

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	<ul style="list-style-type: none">• <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 <p>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:</p> <ul style="list-style-type: none">• January , February , March , April , May , June , July , August , September , October , November , December• 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'. <p>Please note: Two-digit years are expanded to 20xy (so 4/5/82 becomes 4/5/2082).</p> <p>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)</p>
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	<ul style="list-style-type: none">• 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	<p><hour>: <minute>: <second> <am/pm>, e.g. 8:23 pm (becomes 20:23:00) <minute>, <second> and <am/pm> can be omitted.</p> <p>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)</p>
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.

1.5.3 Configuration of the export

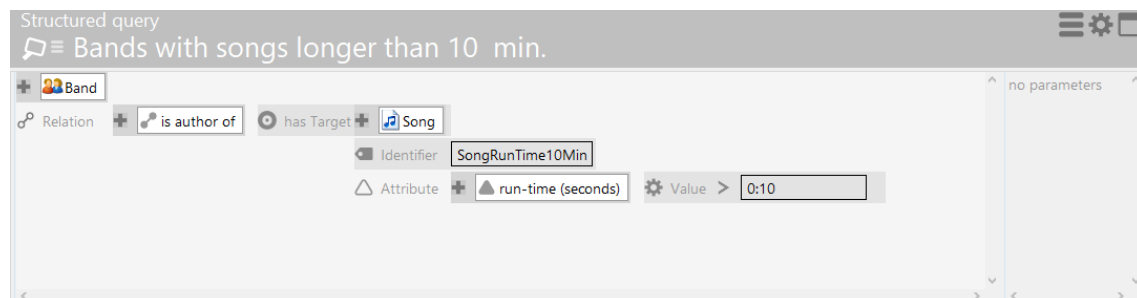
The export of data from a Knowledge Graph into a table is prepared in the same editor and in the same way as the import.

1. A new mapping is created in a table mapping folder in the main window.
2. In the table mapping editor, the file to be generated is specified.

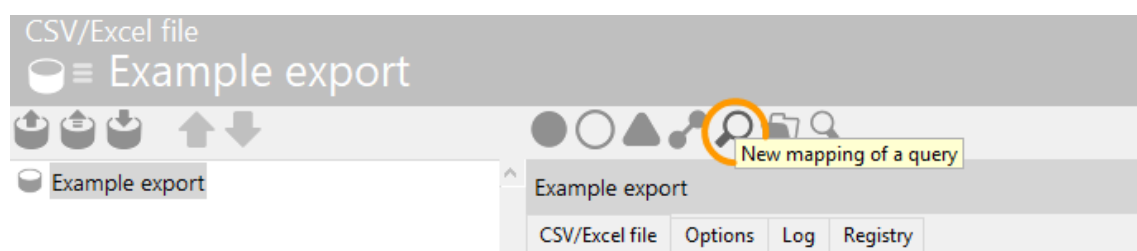
The difference to the import is that the columns are not imported from the table now but have to be created in the table mapping editor. Since the import and export editor are one and the same, you first have to select whether a new column to be created is a *standard* column or a *virtual property*. 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 than 10 min long. To do this, we first have to define a structured query that collects the desired objects.



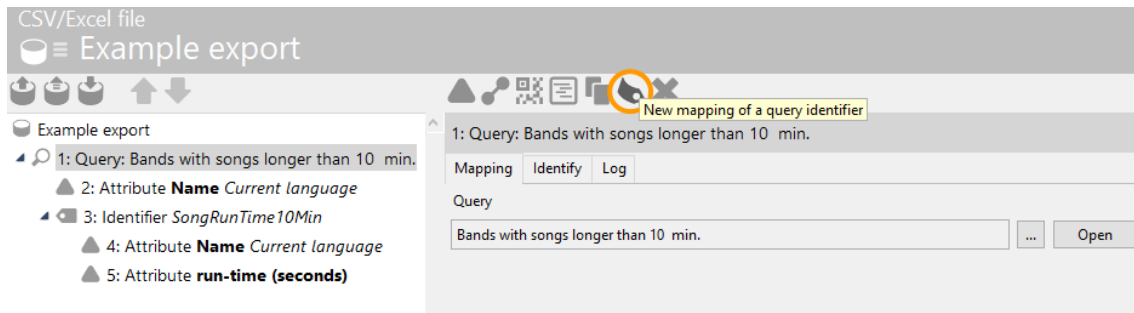
We then access this structured query from the configuration of the export. To do this, we select the mapping of a query rather than an object mapping in the mapping configuration header. The structured query can only be accessed with a registration key.



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

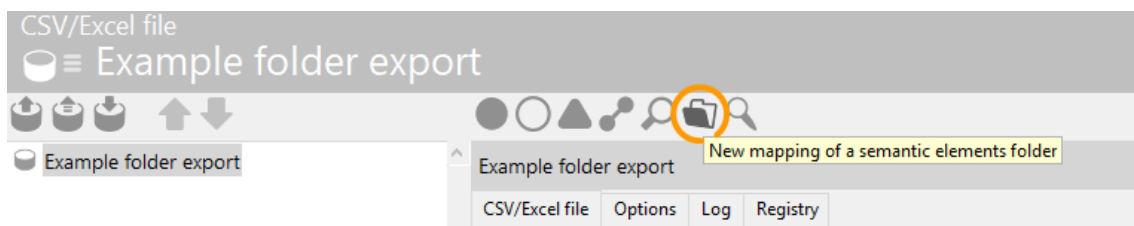


the songs of the bands we have thus compiled but only those songs that also match the search criterion, which is songs longer than 10 min in our example. To do this, we can assign identifiers to the individual search conditions in the structured query. These identifiers in turn can be addressed in the export definition.



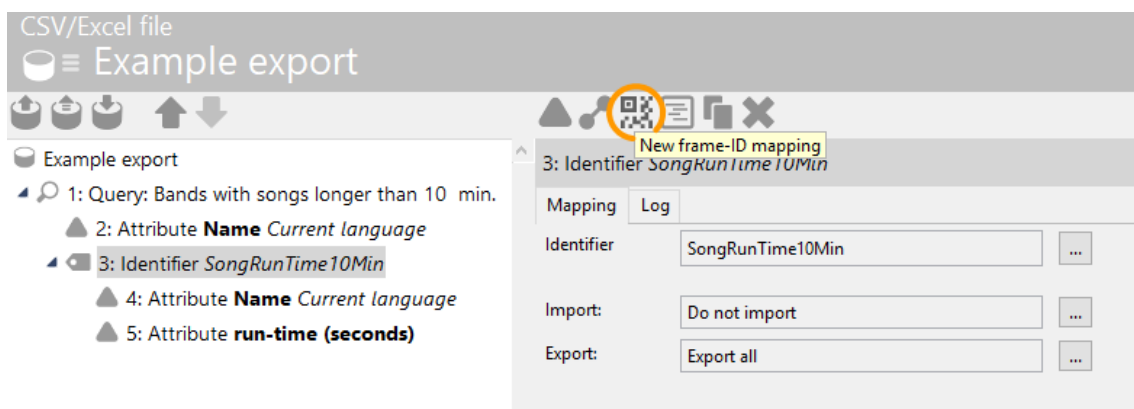
Exporting collections of semantic objects

Collections of semantic objects can also be exported. These also need a registration key, which you can set under TECHNICAL -> Organizing folder.



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:



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 further information on this subject, refer to the "Script mapping" chapter.



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 a DELETE on the entire table followed by an INSERT.

1.5.4 RDF-import and -export

RDF is a standard format for semantic data models. We can use the RDF import and export to exchange data between the semantic graph database and other applications, and also to transport data from one i-views semantic network to another.

During an RDF export, the entire semantic network is exported into an RDF file. RDF import, in contrast, is interactive and selective. That is, we can specify at schema level as well as for individual objects and properties what is supposed to be imported and what not.

Reconciliation from RDF with the existing objects in the semantic network

During import, types and their instances can be identified by means of the following properties:

- **rdf:about**
- **RDF-URI-Alias:** Allows the assignment of different URIs to a semantic element
- **rdf:ID-prefix**
- **rdf:id:** For more information, see <https://www.w3.org/TR/rdf-syntax-grammar/#section-Syntax-ID-xml-base>
- **i-views Frame-ID:** Depending on the settings of the previous export; only applicable if Frame-ID of the element in source network and target network are identical

If the RDF data originates from the same schema as the network into which it is imported, e.g. from a backup copy, the RDF import automatically assigns objects and object types by means of their ID.

There are two possibilities/stages for determining the import mapping of RDF data:

- **Using global settings:** The basic settings allow to determine whether schema is allowed to be changed or not, regardless of the kind of types. Identified objects always are going to be updated, non-identifiable objects are going to be created.
- **Specifying detailed settings manually:** When due to external RDF files a type-dependent



correction of the assignment is needed, both the import strategy and the assignment for each type can be specified manually. Since this manual kind of mapping can be error-prone and exhaustive for extensive RDF files, it is recommended to prefer the import by using global settings and adequate RDF-URI assignment.

If the data originates from another source, the default setting of the import is into a separate subnet. Pay attention that a lossless export and import of metadata structure is not always possible, especially regarding meta relations.

1.5.4.1 Basic principles

In this section we have a look on the basic principles of RDF and the special cases to be obeyed for import mapping. For further information about the RDF standard, see: w3c.org/rdf.

In general, the i-views Knowledge-BUILDER supports XML-RDF.


For identifying the content within the RDF file and the Knowledge-BUILDER as well, the RDF-URI is used. It comprises the base URI (= base URL) and the RDF-ID:

`[RDF-URI] = [Base URI] + [RDF-ID]`

The **Base URI** syntax in RDF is constructed by the "xml:base" prefix, like in "xml:base=http://example.org/". The base "base" is only a namespace for individual domains; the qualifier "xml" is for readability reasons in terms of XML transportation, which is irrelevant for import.

A relative URI in RDF is built up by the syntax "rdf:about". Attribute values are most likely text between tags: `<rdf:prefLabel xml:lang="en">Example</rdf:prefLabel>`, surrounded by the translation layer identified by "xml:lang". Relations will be formed by RDF-entries like "rdf:resource". IDs will only be created via import in the Knowledge-BUILDER if they are literally written in the RDF file. The RDF-ID is no absolute identifying characteristic! It is not recommended to set RDF-IDs manually in the RDF file, since duplicate values can lead to data being mislocated.

Global settings

The Knowledge-BUILDER Base URL is defined in the settings menu  and it is valid for both import and export:

Qualifier	Namespace
iirds	http://iirds.tekom.de/iirds#
schema	http://schema.org/

“Additional namespaces” is for export only.

Note: Always use a local copy of the network for trying out RDF-Import. If all settings led to a successful import, then make the import on the real instance.

Possible issues


- In most cases when importing external RDF (RDF which didn't have been created by the Knowledge-Builder itself out of the same knowledge network), the namespaces possibly won't fit. This results into lots of types within separate main types being created in the network.
Therefore we can prepare the import as described at the end of this section.
- In RDF, the definition and assignment of properties can lead to creation of many objects in the network which normally should be formed into an attribute value of some certain object instead.
Therefore a manual correction of the type assignment in the mapping or an alternative XML import using XPath expressions (XPath 1.0) might be needed.
- Don't choose the option “Identify objects with global URI also by local ID” if the base URL in the RDF differs from the Knowledge-Builder base URL. Furthermore, some RDF-ID in the RDF file could be identical by accident with some existing ID in the network, resulting into the object in the network being overwritten!
Always use the RDF-URI for identification.
- If your RDF file doesn't contain a base URL, the file path of the RDF will be used as the base URL instead.
This can be checked by opening the import dialog first. We then can add the RDF-URI or RDF-URI-Alias accordingly and then check the assignment again by opening the import dialog once again.



Preparation before import

Imports can be prepared regarding type assignment in the case that the RDF files contains foreign base URLs. Because RDF imports can lead to schema changes, it is always recommended to try the RDF import on a local copy of the knowledge-network before. To do so, we continue as follows:

1. Before importing, first create the scheme manually (object types, attribute types and relation types).
2. For the types, add the RDF properties by clicking on "Add attribute or relation" :

 Choose property — □ ×

Choose property

Name	Defined for	Supertype
RDF-URI-Alias	Instances of Top-level type, Types of Top-level type	Attribute
rdf:about	Instances of Top-level type, Types of Top-level type	Attribute
rdf:ID	Instances of Top-level type, Types of Top-level type	Attribute
rdf:ID-Prefix	Types of Top-level type	Attribute

OK Cancel

RDF-URI-Alias: Further attribute, if the element is being fed by several RDF with different URIs

rdf:about: Attribute for "rdf:about"

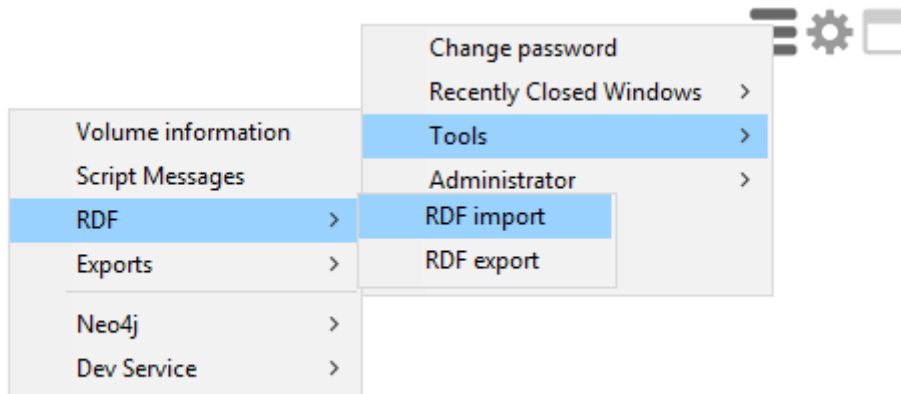
rdf:ID:

rdf:ID-prefix:

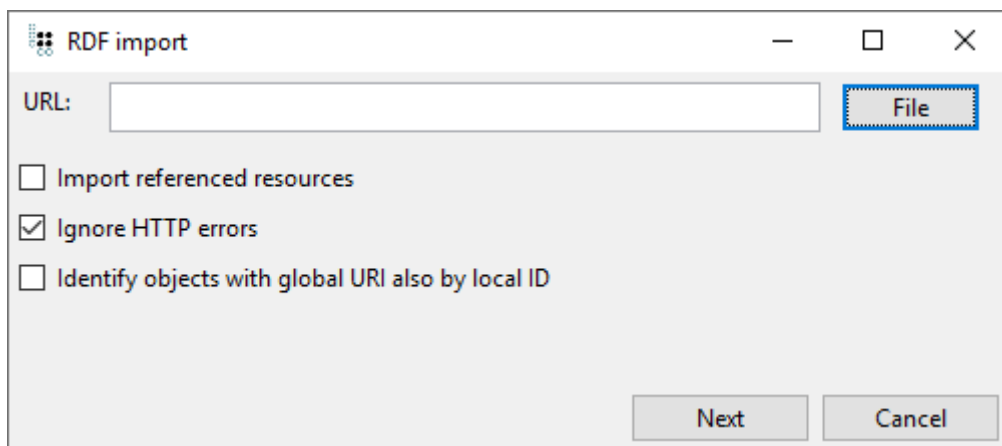
3. Open import dialog and check import mapping.
4. If the adjustments lead to the intended mapping, start the import and check the result.

1.5.4.2 RDF import

For accessing the RDF import mechanism, go to the global actions settings and choose Tools > RDF > RDF-Import.



A dialog opens for choosing the import file:



Options:

- Import referenced resources:**
 If this option is chosen, all referenced resources specified in the RDF file are going to be imported additionally.
 Note: Be aware that the referenced resource itself can contain further references, leading to much more data being imported than initially intended.
- Ignore HTTP errors:**
 The Knowledge-BUILDER will return error messages if the RDF namespace label is missing after the URL; only the namespace http-URL at the top will be considered.
- Identify objects with global URI also by local ID:**
 This option only makes sense if the rdf to be imported is originated from the same knowledge network for which it is intended to be imported. Importing RDF with only considering the ID can lead to data being overwritten when the RDF is from another

domain and the IDs match accidentally. This option does not make sense when the RDF base URL differs from the knowledge network base URL.

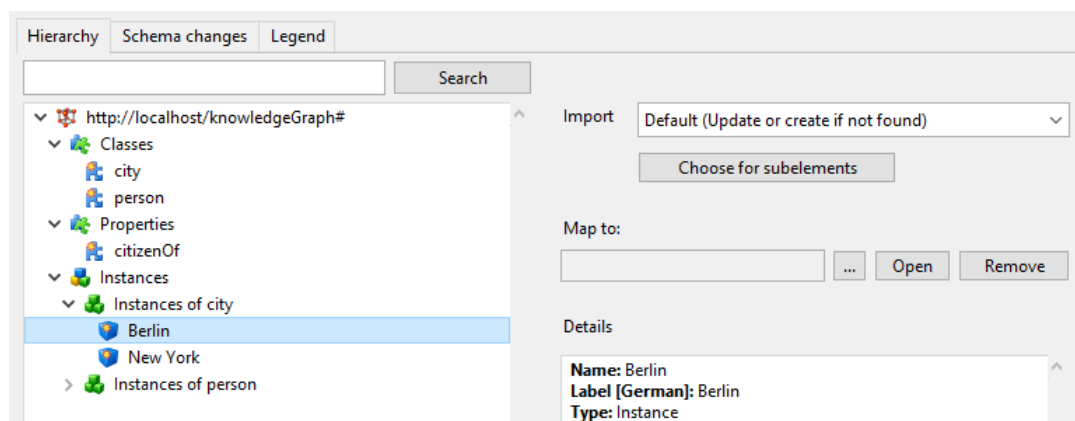
Note: When importing RDF, for every unknown namespace a separate main type will be created in the knowledge network. The assignment of RDF content to dedicated knowledge network types depends on how the information is represented in the RDF file.

Setting the import options

1. Manual supertype mapping:

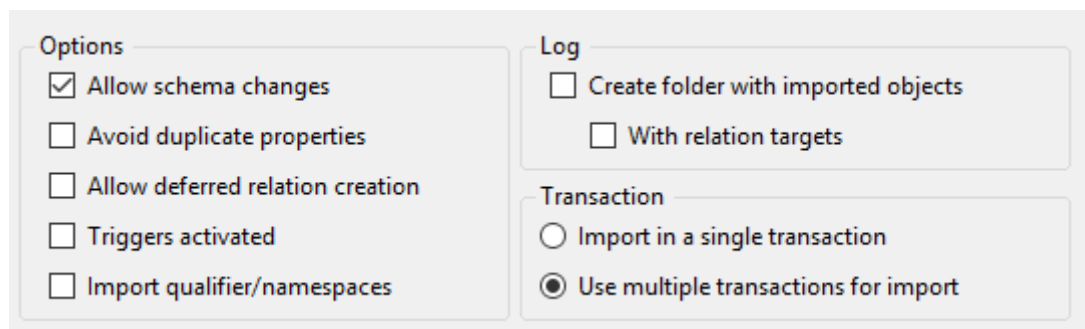
Per default, the RDF-URL (RDF ontology) will be used as supertype assignment.

For every type within the RDF, you can choose the supertype mapping in the semantic network manually if a different type assignment is needed:



In order to be sure about which supertypes will be created by the import, you can check this in the “Schema changes” tab. By clicking on “Show in tree”, you can quickly jump to the location of a type in the hierarchy structure. The “Legend” tab explains the import mapping symbols.

2. Import options:



a. Allow schema changes: Since you don't want a file to change the schema, it is recommended to disable this option

b. Avoid duplicate properties: Because in RDF properties cannot be assigned with an ID, a unique identification of properties and their values within the knowledge network is not possible when importing RDF without krdf. When you want to import a foreign RDF without krdf, it is recommended to enable this option.

When transferring RDF between knowledge networks, knowledge network specific at-



tributes can be identified by means of the enhanced krdf syntax. This includes properties for view configuration, REST configuration, attribute values, relation targets, meta-properties on relations etc. In this second case, it might be needed to disable the option. Pay attention that krdf adds the internal frame IDs for instances and properties whereas external IDs have no impact on identification of such content. The Knowledge-Builder automatically creates unique frame IDs when new elements are created within the knowledge network - either by an import or by the user.

c. Allow deferred relation creation: When importing data from public resources, the attribute "reference to URL" can be created as a substitute reference for (temporarily unavailable) dependencies. The attribute then can be used for re-identification in deferred imports. This might be useful when empty parts including URL without type definition exist within the RDF file.

d. Triggers activated: Normally, triggers are not activated during RDF import. If you nevertheless wish triggers being activated, enable this option.

e. Import qualifier/namespace: This option only makes sense when re-importing RDF that has been previously created out of the same network. If you import a RDF with a foreign namespace, skip this option.

3. Log options:

a. Create folder with imported objects: This option allows you to inspect the imported objects within a folder that will be created in the working folder.

b. With relation targets: When the RDF file contains new objects with relations to targets that already exist in the knowledge network, the relation targets will be included in the folder of imported objects.

4. Transaction options:

a. Import in a single transaction: This is the most common import method.

b. Use multiple transactions for import: This option is recommended when the RDF file contains a huge amount of content or when the connection to the external resource might be weak or unstable. When an error occurs, the amount of content affected by a rollback will be less due to the increased import steps in terms of transactions.

5. If you checked all settings, start the import and check the result.

Alternative: XML import mapping

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. For more information, see chapter "XML Import Mapping".

Further RDF import/export possibilities

RDF files also can be imported or exported via the REST interface by means of a JavaScript mapping. In this case, only global options for import are available as specified in the JavaScript API documentation:

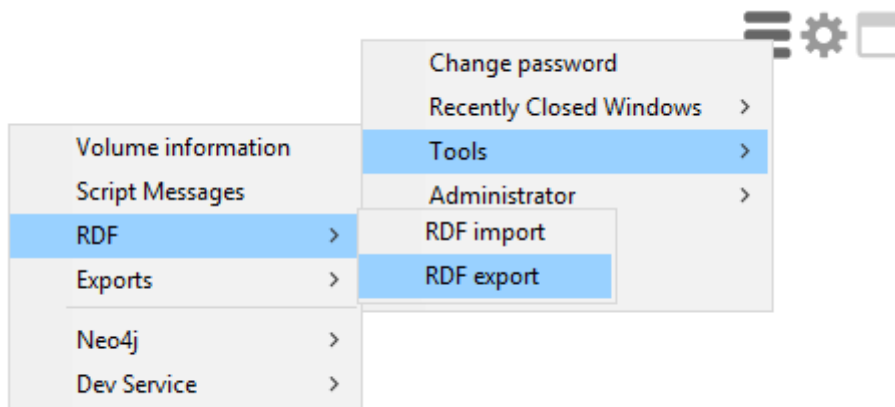
[http://documentation.i-views.com/5.4/javascript-api/\\$k.RDFImporter.html](http://documentation.i-views.com/5.4/javascript-api/$k.RDFImporter.html).

Exceptions: Within i-views content, URIs are generated automatically for the semantic elements when being created in the knowledge network.

1.5.4.3 RDF export

Exporting the whole semantic network as RDF

On the global actions menu, select Tools > RDF > RDF export.



Exporting parts of the network

It is possible to export just a part of the knowledge network, for example:

- Listed elements from an objects list
- Elements from within a semantic elements folder
- Elements from within a graph editor bookmark

If you wish to export listed elements **without collecting** them in a folder:

- Select the list elements to be exported. Open the context menu by means of a right click. Then choose "RDF export".

For **collecting** the elements to be exported, there are several possibilities:

1. Create a semantic network elements folder and add the elements:
 - a. In your private or working folder, create a semantic network elements folder.
 - b. Go to the objects list of your choice and add the elements to the folder by dragging & dropping them.

-or-

Select the elements in the object list and open the context menu by means of a right click. Then choose "Store selected elements in folder".
2. Right-click on the semantic network elements folder and choose "RDF export".

If you wish to put **all selected list elements** into a semantic network elements folder:

1. Open the context menu by means of a right click. Then choose "Copy semantic elements



to new folder”.

2. Right-click on the semantic network elements folder and choose “RDF export”.

If you have already created a **bookmark of a graph editor view**, you simply can export them: Right-click on the bookmark and choose “RDF export”.

Note that only the content of the selection (of the folder or bookmark) will be exported. In terms of an object, this will be the cluster containing the attributes and the relation halves directly attached to the contained object only.

Note: When no base URL is specified in the global settings of the Knowledge-BUILDER, the path name of the RDF export file will be used as base URL instead.

RDF Exporting settings

The screenshot shows the 'RDF Exporting settings' dialog box. It has the following fields and sections:

- File:** A text input field with a blue border and a small '...' button to its right.
- Base URL:** A text input field containing the value 'https://i-views.com/kb#'.
- Qualifier:** A text input field containing the value 'iv'.
- Syntax:** A section with two checked checkboxes: 'Use OWL' and 'Use KRDI'.
- Scope:** A section with five checkboxes: 'Export schema only' (unchecked), 'Export labels' (checked), 'Export meta properties' (checked), 'Export extensions' (checked), and 'Enhanced comments' (checked).
- IDs:** A section with two radio buttons: 'Local IDs (rdf:ID)' (unchecked) and 'Use full URLs (rdf:about)' (checked). Below them are two checkboxes: 'Create attributes for generated URLs and IDs' (checked) and 'Do not use stored URLs and IDs' (unchecked).
- Frame-IDs:** A section with three checkboxes: 'Use frame URLs (krdf:frame)' (checked), 'Export Frame-IDs of types and objects' (checked), and 'Export Frame-IDs of attributes and relations' (unchecked).

Syntax

- **Use OWL:** Since OWL (web ontology language) allows more options than the conventional RDF syntax, this option is always recommended except the case that the RDF is going to be reused for another system which does not accept OWL.
- **Use KRDF:** The KRDF syntax is i-views specific. It allows more enhanced constructions or representations compared to RDF or OWL like the following:
 - o Instances that have several supertypes
 - o Domains that consist of an intersection of supertypes
 - o Frame IDs of semantic knowledge network elements

Scope

Note: The scope options comprise only schema (types) of the whole export

- **Export schema only:** This is a simplified feature for ensuring only to export schema of



the Knowledge Graph and not Instances.

- **Export labels:** If activated, labels will not be exported as an attribute but in forms of a label literal with the syntax `<label xml:lang="eng">`.
- **Export meta properties:** In terms of official RDF specification, meta properties are out of scope. Nevertheless, meta properties can be regarded as a construct with statements about statements, as described in the reification rules of the RDF specification. Therefore, this option is useful when re-importing into an i-views semantic knowledge network is intended.
- **Export extensions:** This option allows the export of extensions of semantic objects.
- **Enhanced comments:** When enabled, XML comments with real name will be created. The exported RDF file will contain comments for dividing up into sections for objects, related objects and referenced schema hereafter, including statements about the relationships from each individual object to the related object.

IDs

- **Local IDs (rdf:ID):** This option only makes sense when re-importing the resulting rdf into the same knowledge network or into a highly similar knowledge network with the same namespace and correct IDs. If the target network accidentally has existing elements with same ID, the elements might be overwritten without further recognition.
- **Use full URLs (rdf:about):** This option should be preferred, since the full RDF URL contains the namespace and thus ensures correct mapping when reimporting the RDF, provided the base URL of both RDF file and settings being identical.
- **Create attributes for generated URLs and IDs:**
- **Do not use stored URLs and IDs:**

Frame-IDs

- **Use frame URLs (krdfframe):** This option is only available in combination when used with full URLs instead of IDs. It provides internal URLs built up by frame IDs of the semantic knowledge network elements additionally.
- **Export Frame-IDs of types and objects:** Exporting frame IDs only is useful in the case if duplicating parts of the existing network is intended. Since frame-IDs change in various cases and differ highly due to their randomized creation (2^{29} possible values), they cannot be used for another knowledge network.

Frame-IDs keep the same when:

- o Changing the type of an instance
- o Downloading a network
- o Updating a network

Frame-IDs change when:

- o Changing relations into single-sided relations
- o Another instance of knowledge network is used
- o Creating objects, even if they will be given identical properties

- **Export Frame-IDs of attributes and relations:** Exports frame-IDs of properties (attributes and relations) as well. As for exporting frame-IDs of objects and types, this option is useful for (partial) duplicating networks, but not for reuse into foreign networks



1.5.5 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:

1. Open the backup Knowledge Graph in the Knowledge Builder
2. 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.
3. Open the RDF export on the newly created folder using the context menu
4. Specify a file name in the export dialog, select the options "Use URLs (rdf:about)" and "Use frame URLs (krdf:krdf:)" and execute the export:

File: C:\User\Desktop\export.rdf

Base URL: http://localhost/test#

Qualifier: export

Syntax

- ☒ Use OWL
- ☒ Use KRDI

Scope

- ☐ Export schema only
- ☒ Export labels
- ☒ Export meta properties
- ☒ Export extensions
- ☐ Enhanced comments

IDs

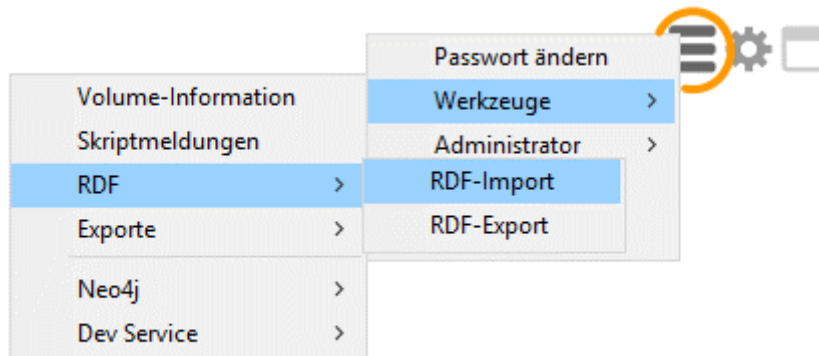
- ☐ Local IDs (rdf:ID)
- ☒ Use full URLs (rdf:about)
- ☐ Create attributes for generated URLs and IDs
- ☐ Do not use stored URLs and IDs

Frame-IDs

- ☒ Use frame URLs (krdf:krdf:)
- ☒ Export Frame-IDs of types and objects
- ☐ Export Frame-IDs of attributes and relations

Note: the option "Use KRDF" results in i-views additionally copying specific content that cannot be mapped in full by means of RDF syntax.

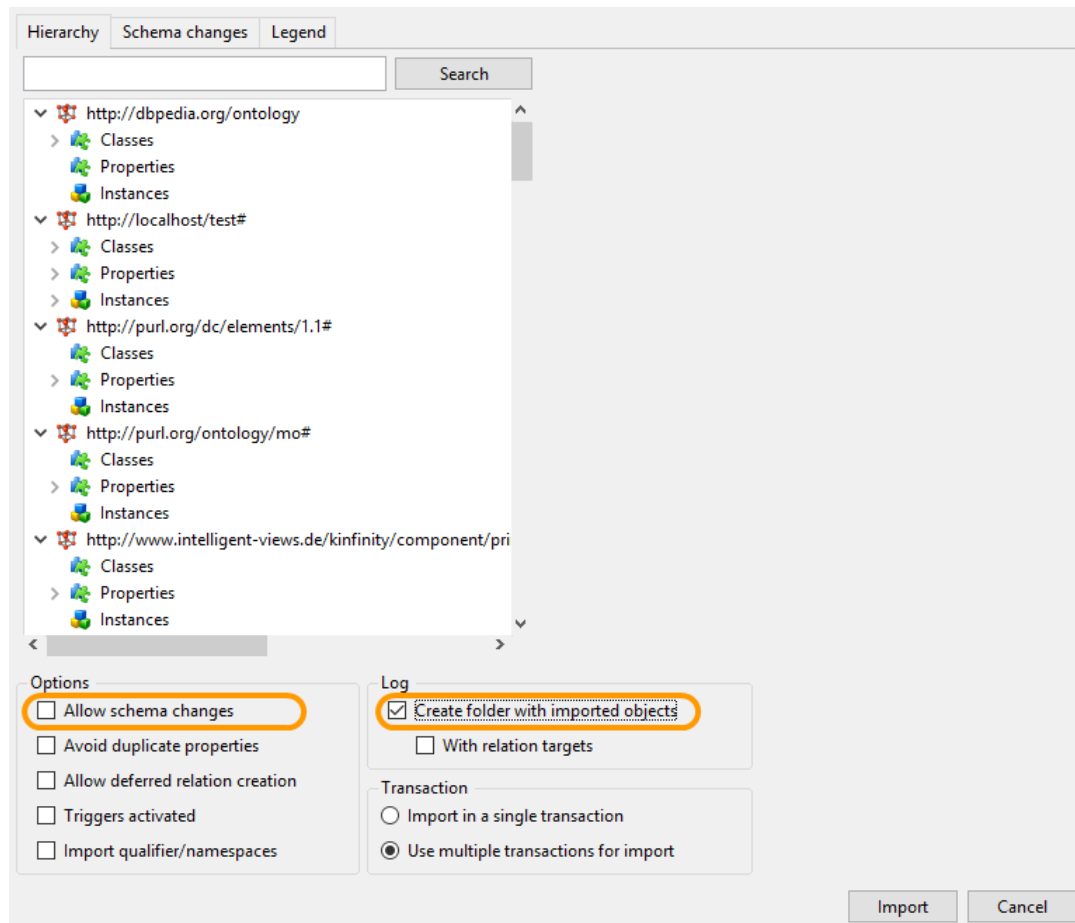
5. Close the Knowledge Builder and open the target graph in the Knowledge Builder
6. Open the RDF import dialog in the main menu under Tools > RDF > RDF import:



7. Select the file and press "Next":



8. Deactivate the option "Allow changes to the schema" in the selection dialog, and activate "Create folder with imported objects":



9. Execute import
10. Check the restored individuals

1.5.6 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.