Semantic Technology:

Machine Reasoning (query-based) +

Some more queries +  
Lucene searches

**Objective:**

* To run simple cases of **Machine Reasoning** (using rules expressed in the SPARQL language)
  + To distinguish between "Reasoning on the fly" (when results are not written to the database) and "Reasoning with writing" (when results are stored in the database)
  + To perform some simple "social network analysis" as far as the query language allows
* To learn **some additional query type**s
  + Aggregation queries
  + Subqueries
  + Queries with results exclusion
* To learn how to extend the query language with a **Lucene search engine**

**Prerequisite:**

* You should be familiar with the queries that were discussed in the previous tutorial (SELECT, DESCRIBE)
* We use the same graph database as in the previous tutorial. We replicate the dataset here for your convenience

@prefix : <http://buchmann.ro#>.

@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

:mymoviegraph

{:JamesCameron :directorOf :Avatar,:Terminator;

:hasName "James Cameron".

:JohnMcT :directorOf :Predator, :DieHard;

:hasName "John McTiernan";

:birthInfo **\_:birthdetails.**

**\_:birthdetails** **:birthDate "1951-01-08"^^xsd:date;**

**:birthPlace :SUA.**

:McG :directorOf  **\_:somemovie**;

:hasName "Joseph McGinty";

:hasNickname "McG".

:SamWorth :hasName "Sam Worthington";

:playedIn **\_:somemovie**;

:playedTheRole **[:asCharacter :JakeSully;**

**:inMovie :Avatar]**.

:Arnold :playedIn :Terminator,:Predator;

:wasGovernorOf :California;

:hasName "Arnold Schwarzenegger";

:birthInfo **[:birthDate "1947-07-10"^^xsd:date;**

**:birthPlace :Austria].**

:LindaH :hasName "Linda Hamilton";

:playedTheRole **[:asCharacter :SarahConnor;**

**:inMovie :Terminator]**.

:ShBlack :playedIn :Predator;

:hasName "Shane Black";

:directorOf :KKBB.

:TimBurton :directorOf :Batman;

:hasName "Tim Burton";

:birthPlace :SUA.

:JackNicholson :playedIn :Batman;

:hasName "Jack Nicholson".

:Batman :hasTitle "Batman".

:Terminator :hasTitle "Terminator";

:hasBudget 6400000 .

:Avatar :hasTitle "Avatar";

:hasBudget 237000000 .

:DieHard :hasTitle "Die Hard".

:Predator :hasTitle "Predator";

:hasBudget 18000000 .

:KKBB :hasTitle "Kiss Kiss Bang Bang";

:hasBudget 15000000 .

\_:somemovie :hasBudget 200000000 .

:JakeSully :hasName "Jake Sully".

:SarahConnor :hasName "Sarah Connor".

:California :hasName *"California"@en, "Californie"@fr*.

:Austria :hasName *"Austria"@en, "Autriche"@fr*.

:SUA :hasName *"United States of America"@en, "Etats Unis d'Amerique"@fr.*}

## Run your first reasoning rules

**CONSTRUCT vs. INSERT ("Reasoning on the fly" vs. "Reasoning with writing")**

We've already seen that DESCRIBE queries return subgraphs (subsets of full statements). Another query type that does the same is CONSTRUCT, however it works slightly different. Compare the following examples:

DESCRIBE ?x

WHERE

{:Arnold ?relationship ?x}

*=> returns the subgraph having all statements that contain ?x (i.e. all statements about all things to which Arnold has some relationship)*

CONSTRUCT

WHERE

{:Arnold ?relationship ?x}

*=> returns the subgraph of all statements about Arnold (i.e. having Arnold as a subject)*

Therefore:

* DESCRIBE looks for a variable and returns one subgraph for each of its values (it can also look for a single node and return the subgraph around that node)
* CONSTRUCT does not look for a variable, it looks directly for a subgraph pattern

The key benefit of CONSTRUCT is that it can apply transformations "on-the-fly" before returning the subgraph, thus acting like a **Reasoning Rule**. This happens when CONSTRUCT is used with **two patterns**:

CONSTRUCT {?x a :Director. ?y a :Movie} # pattern to be generated

WHERE {?x :directorOf ?y} # pattern to be searched

The example generates two entity types: *:Director* and *:Movie,* for all things connected by the relationship *:directorOf* (it will return a subgraph containing only the generated type declarations). Notice the two query patterns:

* one to be searched, after WHERE (it will select a subgraph);
* plus a transformation pattern, after CONSTRUCT (it generates a new graph, which may contain new information, here the types).

Reasoning rules can also generate relationships – e.g. a collaboration relationship between actors and directors who worked on the same movie, considering that the actor-movie connection has two variants:

CONSTRUCT {?x :workedWith ?y}

WHERE {?x (:playedIn|(:playedTheRole/:inMovie))/^:directorOf ?y}

One key aspect is that CONSTRUCT returns the generated graph without saving it in the database! This is because CONSTRUCT, just like SELECT or DESCRIBE, is considered a read operation – i.e. the generated information is "temporary" (made available only to the client that executed the CONSTRUCT).

* This makes CONSTRUCT useful for **ETL pipelines on read-only graphs** (e.g. in public RDF databases such as DBPedia) – we extract what we need, apply some transformation on the way and deliver the generated stuff to a client;
* For read-write databases we should also be able to save the generated information, to make it available to all future client requests and queries! For example the entity types we've just generated (Director, Movie) will definitely be useful on long term. In order to apply **Reasoning with writing** we have to run an INSERT... WHERE ... operation which looks very similar to CONSTRUCT:

INSERT {?x a :Director. ?y a :Movie}

WHERE {?x :directorOf ?y}

Remember the difference:

* CONSTRUCT returns the generated graph to the client, without storing it (the client app can later choose to save it with a separate operation);
* INSERT stores the generated graph, without returning it (the client app must perform a SELECT to retrieve what was generated).

Both are useful in different scenarios – do we care more about the size of the database, or about the performance of queries? do we want the database to grow or is the generated information useful only to a handful of potential clients?

So INSERT makes the reasoning results permanently available - it now becomes easier to obtain the list of directors in subsequent queries:

SELECT ?x

WHERE {?x a :Director}

Or, if someone never heard about Terminator, they can now find out easily that it is a movie:

SELECT ?x

WHERE {:Terminator a ?x}

Remember that we had difficulties obtaining all budgets and titles, since those properties were missing for some movies (and we were not sure what property "makes a movie be a movie"). Now we can collect movies *by their type*, before applying the OPTIONAL clause on potentially missing properties:

SELECT ?IDfilm ?title ?budget

WHERE

{

**?IDfilm a :Movie.**

OPTIONAL {?IDfilm :hasTitle ?title}

OPTIONAL {?IDfilm :hasBudget ?budget}

}

Remember that we had an anonymous movie (no ID and no title), uploaded under the name \_:somemovie. We've suggested earlier that anonymous node names cannot be used in queries, i.e. the following query will NOT be able to find the budget of that "unknown" movie:

SELECT ?buget

WHERE {**\_:somemovie** a :Movie; :hasBudget ?buget}

However, a query can benefit from the position of the anonymous node – i.e. we cannot ask *What is the budget of \_:somemovie?*, but we can ask *What is the budget of the movie that involved both McG (as director) and SamWorth (as actor)?*

Here is the query in two versions:

* The Turtle-like version shows intuitively how that anonymous node keeps all details connected:

SELECT ?budget

WHERE

{

:McG :directorOf **?anonymous**.

:SamWorth :playedIn **?anonymous**.

**?anonymous** :hasBudget ?budget.

}

* A more concise form makes use of [....], keeping in mind that SamWorth can be reached by navigating in reverse from the anonymous node:

SELECT ?budget

WHERE

{

:McG :directorOf [^:playedIn :SamWorth; :hasBudget ?budget].

}

*Note: We might be tempted to also try the following (which gives correct results on the current data!):*

SELECT ?budget

WHERE

{

:McG :directorOf/:hasBudget ?budget.

:SamWorth :playedIn/:hasBudget ?budget.

}

*In the general case, if multiple movies have the same budget we will get unintended results – i.e. we will get movies that have the same budget (by coincidence), not movies where those two individuals worked together! In this form, the two lines do not necessarily refer to the same node!*

*Beware of this kind of error, which occurs when a query must pass through an anonymous node coming from different directions and fails to specify that it should be one and the same anonymous node.*

**Use cases for Machine Reasoning**

**We saw one key use case for reasoning: to generate types of things, based on the properties of those things!** Moreover, reasoning rules may generate multiple types for the same individual, based on more complex patterns. The next example generates the type :AmericanDirector for those who are directors and are born in USA:

INSERT {?x a :AmericanDirector }

WHERE {?x a :Director; :birthInfo**?**/:birthPlace :SUA}

Notice that we used character ? to indicate that for some individuals (TimBurton) the property :birthInfo is missing and the birth place is directly connected. The next query now can collect all american directors:

SELECT ?x

WHERE

{ ?x a :AmericanDirector }

**Next, we will see the second use case for reasoning: to generate shortcut relationships, whenever a chain of relationships ("multihop relation") has a precise meaning.**

We've mentioned that with reasoning rules we can solve some irregularities in the graph – e.g. to ensure that the relationship between actors and movies is always described in the same way. Run the following reasoning rule:

INSERT {?x :playedIn ?y }

WHERE {?x :playedTheRole/:inMovie ?y}

Basically this generates a shortcut edge, directly connecting actors and movies that originally had an intermediate anonymous node between them (to also connect the character). We can benefit from the generated shortcuts in many ways:

* to find all actor-movie pairs without resorting to a logical OR:

SELECT ?idactor ?idfilm

WHERE {?idactor :playedIn ?idfilm}

* to find all actors and, if available, their characters (missing in some cases); now :playedIn gives us all actors, whereas :playedTheRole gives us the alternative path that leads to the character/role:

SELECT DISTINCT ?actor ?role

WHERE

{

?actor ^:hasName/:playedIn ?idFilm

**OPTIONAL** {?actor ^:hasName/:playedTheRole/:asCharacter/:hasName ?role}

}

To avoid relying on :playedIn just to find actors, we can also generate an entity type for them just as we did with directors:

INSERT {?x a :Actor. ?y a :Movie}

WHERE {?x :playedIn ?y}

*We also regenerated the Movie type, in case some movies were not caught by the director's involvement. For movies that already got the type declaration from the previous rule, there's no problem - GraphDB will only keep one copy that is uploaded or generated multiple times!*

Be careful not to generate invalid statements – e.g. statements that start with a data value, which would happen when trying to execute the rule:

INSERT {?numeActor :workedWith ?numeRegizor}

WHERE {?numeActor ^:hasName/:playedIn/^:directorOf/:hasName ?numeRegizor}

**Finally, we learn a third use case for reasoning: generation of anonymous nodes, to which we plan to attach some information in the future.** For example, we can insert an anonymous node between things that have a direct connection.

With the next example we will generate an intermediate node between Tim Burton and his birth place – this will make sure that everyone will have an anonymous node between their ID and their birth place (to potentially attach the birth date, as we have it for Arnold):

INSERT

{?thing :birthInfo [:birthPlace ?place]}

WHERE

{

?thing :birthPlace ?place.

FILTER (isIRI(?thing))

}

To conclude, we exemplified three common reasoning use cases:

* **generation of entity types** <--- from the relationships where individuals are involved;
* **generation of shortcut edges** <--- from a relationship chain of some particular meaning;
* **insertion of anonymous nodes ("breaking" relationships[[1]](#footnote-1))** <--- where we foresee the future need to group multiple properties around an anonymous node.

*Of course, the three techniques can be combined (even in the same INSERT rule) for more sophisticated patterns.*

*However, it is often the case that simple reasoning rules are executed in a step-wise manner like we did here – each rule relying on the results of the previous one (it lowers the risk of generating unintended information, it gives us better control)*.

## Analyze a social network constructed with Machine Reasoning

**Derive and visualize the social network**

First, we use reasoning to generate a special kind of "shortcut": a **direct "collaboration" relationship** between all actors who worked on the same project.

* Turtle-like variant:

INSERT {?x :workedWith ?y }

WHERE

{

?x :playedIn ?film.

?y :playedIn ?film.

FILTER (?x!=?y)

}

* Path-like variant:

INSERT {?x :workedWith ?y }

WHERE

{

?x :playedIn/^:playedIn ?y

FILTER (?x!=?y)

}

Notice in both variants the FILTER, which will avoid generating the new relation between an actor and himself (this is a risk, since the query will walk each *:playedIn* relation in both directions).

Next, we generate the same relationship *between directors and actors that worked on the same movie* – now FILTER is not necessary because we don't have cases where the same person was both actor and director for the same movie (in reality this case should also be considered)

INSERT {?x :workedWith ?y }

WHERE

{

?x :directorOf/^:playedIn ?y

}

*These last couple of INSERTs are actually enriching our graph with a* ***"social network" component*** *– i.e. people who know each other because they were involved in common projects (including those involved in the unknown \_:somemovie). This is a proof of how Machine Reasoning can reveal information not initially expressed by the database. If this example would not be about movies, it could be an approach to detect conflicts of interest or suspicious fraud circles.*

We can display now all pairs of persons who collaborated (their names):

SELECT ?x ?y

WHERE

{

?x ^:hasName/:workedWith/:hasName ?y

}

Or, we can isolate the social network through "subgraph extraction" – i.e. by running a CONSTRUCT:

CONSTRUCT

WHERE

{

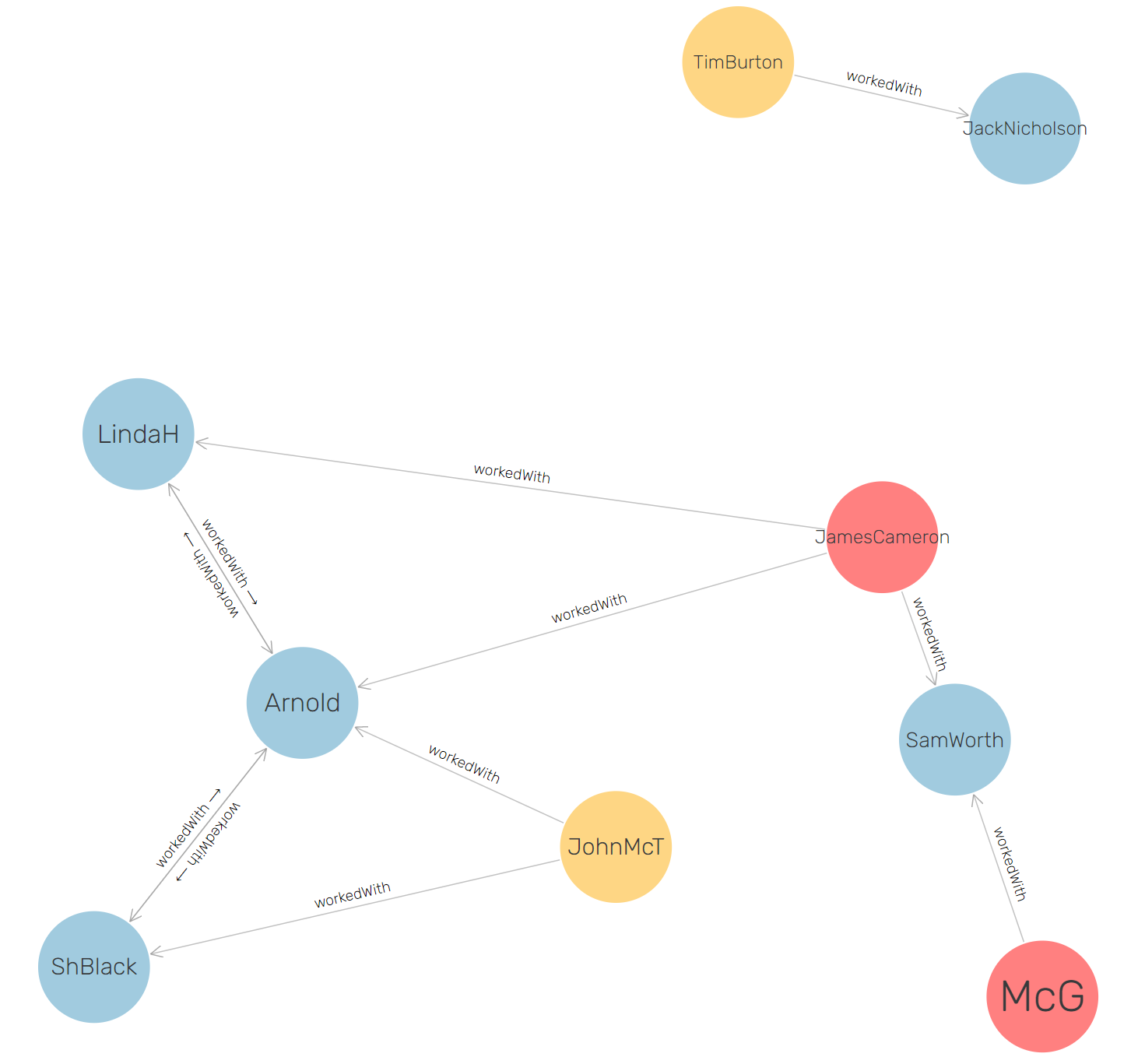
?x :workedWith ?y.

?x :hasName ?namex.

?y :hasName ?namey.

}

Because CONSTRUCT returns a subgraph, when results are displayed we can use the *Visual* option to visualize the social network:



Unlike SELECT, a CONSTRUCT does not simply return an array but a subgraph – which means we can use the *Download* button to isolate this social network in a separate file, if we need to analyze it separately from the rest of the graph database. Or, if we send the CONSTRUCT query from a client app (through HTTP), the social network will be delivered to the client app which may want to apply some graph analytics on it[[2]](#footnote-2).

*This is an Artificial Intelligence behaviour – an app extracting a social network from another app's data through deductive reasoning (although no social information was available in the original dataset!). Machine Reasoning is actually the complement to Machine Learning, both being building blocks of Artificial Intelligence (i.e. the ability to learn, plus the ability to reason on what is learned).*

Notice the obtained graph structure:

* Collaborations between actors and actors are bidirectional (generated both ways, between the two ends of the symmetric path :playedIn/^:playedIn)
* Collaborations between directors and actors are single-direction (from directors to actors, since the generation path was :directorOf/^:playedIn)

*We could adjust the graph to have the collaboration relationship expressed in the same way everywhere. We can choose between:*

* *Having it bidirectional everywhere – queries will be simpler and faster but the database size will grow;*
* *Having it unidirectional everywhere – the database will be kept minimal, but queries will have to navigate collaborations in both directions.*

**Analyze the social network through queries**

We leave the graph as is to get a better feeling of implications. Generally we will NOT rely on bidirectional collaboration edges.

For example, we want to obtain all collaborators of Arnold, regardless of what they are. We must consider **both incoming and outgoing edges of Arnold**  to make sure that we cover everyone. This implies the need to use a logical OR in the query:

* Path-like syntax:

SELECT DISTINCT ?y

WHERE

{

:Arnold :workedWith|^:workedWith ?y

}

* Turtle-like syntax:

SELECT DISTINCT ?y

WHERE

{

{:Arnold :workedWith ?y}

**UNION**

{?y :workedWith :Arnold}

}

Next, we exemplify a variety of queries reflecting typical "social network analysis" investigations:

* Get everyone who is at "one degree of separation" from Arnold (i.e., two hops away from Arnold); we have to FILTER out Arnold because some double-hop connections will return to him:

SELECT DISTINCT ?x

WHERE

{

:Arnold (:workedWith|^:workedWith)/(:workedWith|^:workedWith) ?x

FILTER(?x!=:Arnold)

}

* Get all **common collaborators** of *:Arnold* and *:SamWorth*:

SELECT DISTINCT ?x

{

:Arnold (:workedWith|^:workedWith) ?x.

?x (:workedWith|^:workedWith) :SamWorth.

}

* Get the **full social network of Arnold** – i.e. all individuals that can be reached on a social path of any length. The multiplier **+** (=one or more) can be used to navigate multiple hops of the same relationship[[3]](#footnote-3):

SELECT DISTINCT ?x

WHERE

{

:Arnold (:workedWith|^:workedWith)**+** ?x

FILTER(?x!=:Arnold)

}

*Be careful when using multipliers – they will not return only the nodes found at the end of the path (as we are used from other "graph path" queries), they will return everything found at every intermediate step, by gradually extending the path:*

* *Nodes found at distance of one connection,*
* *Nodes found at distance of two connections*
* *... etc. until the relationship cannot multiply itself anymore.*

Notice that the results do not include Tim Burton and Jack Nicholson, who (as seen in the visualization) are isolated from Arnold's social group (no chain of social connections can reach them!).

Other examples:

* Get **movies directed by everyone who belongs to Arnold's social network** – multiplier + can be followed by whatever properties we need to collect along the extending path (i.e., directed movies found at every hop as we walk away from Arnold on social connections):

SELECT DISTINCT ?x

WHERE

{

:Arnold (:workedWith|^:workedWith)+/:directorOf ?x

}

* Get **people with whom Arnold forms a 3-clique** (a 3-clique is a "gang" of three persons who all know each other):

SELECT DISTINCT ?x ?y

WHERE

{

:Arnold (:workedWith|^:workedWith) ?x.

?x (:workedWith|^:workedWith) ?y.

?y (:workedWith|^:workedWith) :Arnold.

}

(every clique is detected twice because each connection is navigated in both directions)

* Ask **if there is a direct social connection** between Arnold and McG – we get a negative response:

ASK

{

:Arnold (:workedWith|^:workedWith) :McG

}

(reminder: ASK returns a boolean, it only checks for the existence of patterns).

* Ask **if there is a direct chain of social connections** between Arnold and McG (if both are nodes in the same social network) – we get a positive result:

ASK

{

:Arnold (:workedWith|^:workedWith)+ :McG.

}

* However, between Arnold and TimBurton there's not even a chain of social connections:

ASK

{

:Arnold (:workedWith|^:workedWith)+ :TimBurton.

}

* Knowing that a chain of social connections exist between Arnold and McG, see if there's a single intermediate node and return it:

SELECT ?x

{

:Arnold (:workedWith|^:workedWith) ?x.

?x (:workedWith|^:workedWith) :McG.

}

* There's no single node, so we try with two:

SELECT ?x ?y

{

:Arnold (:workedWith|^:workedWith) ?x.

?x (:workedWith|^:workedWith) ?y.

?y (:workedWith|^:workedWith) :McG.

}

Now we find that Arnold can reach McG through JamesCameron and SamWorth. We could group the two attempts in a single query by applying a logical OR to test multiple lengths of the social path.

SELECT ?x ?y # if we check more lengths, we will add more variables

{

{

:Arnold (:workedWith|^:workedWith) ?x.

?x (:workedWith|^:workedWith) :McG

}

UNION

{

:Arnold (:workedWith|^:workedWith) ?x.

?x (:workedWith|^:workedWith) ?y.

?y (:workedWith|^:workedWith) :McG

}

UNION

{

# we can add here longer chains

}

}

*Although SPARQL allows us to detect if there is a path between two things, it does not tell us how long is that path, and it cannot return distinct paths - at least not in the current standard, version 1.1. This can be handled in several ways:*

* *Some servers offer extensions to the standard – e.g. Stardog adds a PATH query*[[4]](#footnote-4) *which offers a multitude of path management operations. This is not currently supported by GraphDB, which chooses to be standard-compliant (the next SPARQL version was announced in 2019*[[5]](#footnote-5)*, currently work in progress*)*;*
* *Whatever we cannot do with the query language we can delegate to the client app code. For example if our social network is queried from Python, it can be easily converted to a Python graph with libraries such as NetworkX*[[6]](#footnote-6) *- a strong API for analyzing network structures, much more powerful than what can be done with a query language;*
* *Although SPARQL 1.1 cannot return individual paths, it can return all intermediate nodes or edges – this is helpful to obtain paths in unidirected graphs (e.g. trees) where we know there are no multiple paths between two nodes. In our case however, since cycles/loops are present in the graph, trying to get all intermediate nodes will actually gives us the entire social network (loops will be circulated in all possible directions). We can check this by applying the multiplier +:*

*SELECT DISTINCT ?x*

*{*

*:Arnold (:workedWith|^:workedWith)+ ?x.*

*?x (:workedWith|^:workedWith)+ :McG.*

*}*

*We can try however such queries in those parts of the graph that contain no cycles.*

**Wildcard paths (arbitrary length chains of arbitrary relations)**

The next example retrieves intermediate nodes between McG and JakeSully, knowing that there are three connections between them:

SELECT ?relation1 ?x ?relation2 ?y ?relation3

{

:McG ?relation1 ?x. ?x ?relation2 ?y. ?y ?relation3 :JakeSully.

}

We would like to find these connections without knowing how many they are, but SPARQL 1.1 has a syntactic restriction – path-like queries cannot have intermediate variables, i.e. it's not possible to run:

:McG ?relation1/?relation2/?relation3 :JakeSully.

Which means it's also not possible to have a "wildcard path" such as:

:McG (?relation)+ :JakeSully.

The good news is that SPARQL 1.1 does have a placeholder (wildcard) for "any relation":

(:|!:)

And if we apply multipliers to it (\* or +) we create a directed chain of "any relations" having arbitrary length (the relation can vary along the way, unlike the case when the multiplier is applied to a specific relation):

(:|!:)\*

With this placeholder we have more possibilities to analyze graphs with the query language:

* **All intermediate nodes** from McG to JakeSully (the result will be SamWorth and an anonymous node):

SELECT ?intermediateNode

{

:McG (:|!:)+ ?intermediateNode.

?intermediateNode (:|!:)+ :JakeSully.

}

* **All intermediate nodes as well as intermediate relation** (the full path, without knowing how long it is – we can calculate that also with a COUNT on intermediate nodes).

SELECT ?intermediateSubject ?intermediateRelation ?intermediateObject

{

:McG (:|!:)\* ?intermediateSubject.

?intermediateSubject ?intermediateRelation ?intermediateObject. **#this is a sliding window!**

?intermediateObject (:|!:)\* :JakeSully.

}

As mentioned earlier, this gives us a full path only if we know that there is a single path. Otherwise we get a mix of all intermediate nodes and relations. Such a mix could be extracted with a CONSTRUCT to be further analyzed separately:

CONSTRUCT { ?intermediateSubject ?relation ?intermediateObject }

WHERE

{

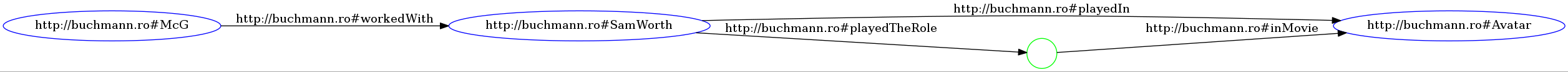
:McG (:|!:)\* ?intermediateSubject.

?intermediateSubject ?relation ?intermediateObject.

?intermediateObject (:|!:)\* :Avatar.

}

If you download the CONSTRUCT result and upload it in a visualization tool (<http://www.ldf.fi/service/rdf-grapher>) you can easily see the multiple paths.



But of course, in order to analyze the paths automatically this result should be passed to a graph analytics library such as the previously mentioned NetworkX for Python.

If we are already aware of certain paths we can, for example, filter paths based on if they are passing through certain nodes or not – e.g. we can obtain a single path here if we avoid the one passing through the anonymous node:

SELECT ?intermediateSubject ?relatieIntermediara ?intermediateObject

{

:McG (:|!:)\* ?intermediateSubject.

?intermediateSubject ?relatieIntermediara ?intermediateObject.

?intermediateObject (:|!:)\* :Avatar.

**FILTER(isIRI(?intermediateSubject)&&isIRI(?intermediateObject))**

}

*We summarize here the path management limitations (unless we work on a server like StarDog which provides dedicated PATH queries):*

* *We can obtain a path if we know its length;*
* *We can obtain all intermediate nodes and relations without knowing how many they are (but we cannot group them on paths) – this will give us a path of any length if there are no multiple paths.*

## Aggregate calculations, subqueries, excluding query results

**Aggregate calculation**

Just like in SQL, we have several aggregate functions – SUM, COUNT, AVG etc.[[7]](#footnote-7)

We typically use them in two ways:

* **Simple aggregates**, when the query returns a single aggregate value – e.g. the total budget:

SELECT (SUM(?budget) AS ?totalBudget)

{

?film :hasBudget ?budget

}

* **Group-level aggregates**, when we use the clause GROUP BY and separate aggregate calculations by group. The next example returns directors together with their average budget and the number of movies for each (we simplify by taking only movies that have a budget declared[[8]](#footnote-8)):

SELECT ?director (**AVG(?budget)** AS ?averageBudget) (**COUNT(?film)** AS ?movieCount)

{

?director :directorOf ?film.

?film:hasBudget ?budget.

}

**GROUP BY ?director**

**HAVING (?averageBudget>15000000)**

**ORDER BY ?averageBudget**

Notice some rules for correctly aggregating at group level:

* The clause GROUP BY should indicate variables (can be more than one) that are returned by the query but are not aggregated – here, only the director ID;
* The variables returned by the query can be of two kinds: the grouping criteria (in GROUP BY) and all the others should be aggregated variables;
* Optionally we may add the clauses ORDER BY (to sort results) and HAVING (to filter groups)

Other examples:

* On the social network discussed earlier, order all individuals by the number of collaborators they have – in some cases collaborators will be counted twice (because of some bidirectional relationships), so we also use DISTINCT to avoid counting twice:

SELECT ?person (**COUNT(DISTINCT ?collaborator)** AS ?collabCount)

WHERE

{

?person :workedWith|^:workedWith ?collaborator

}

GROUP BY ?person

ORDER BY ?collabCount

* We group again movies by director, but now for each director we build a string that concatenates a list of their titles (GROUP\_CONCAT with comma as a separator):

SELECT ?director (**GROUP\_CONCAT(?title;separator=",") AS ?movieList**)

WHERE

{

?director :directorOf/:hasTitle ?title

}

GROUP BY ?director

**Subqueries**

Subqueries are needed in complex operations, in use cases such as:

* Chaining multiple queries (one query must take input from the previous query output)
* A non-SELECT operation (delete, insert) must perform aggregate calculations, which are only possible in SELECT queries
* Multiple successive aggregate calculations must be performed, but cannot be done in a single SELECT
* a SELECT must exclude the results detected by another SELECT

*Example 1. Chaining queries.*The next example retrieves the director of the movie having the smallest budget – we first have to find which is that budget, and only then to look for the director (so we have chained queries):

SELECT ?director **?minBudget**

WHERE

{

?director :directorOf/:hasBudget **?minBudget**

{

SELECT (**MIN(?budget) AS ?minBudget**)

WHERE {?film :hasBudget ?budget}

}

}

The variable ?minBudget is a result of the inner query and must be passed as input to the main query (it is also displayed besides the director ID).

*Example 2. Aggregate calculations in reasoning rules*. Subqueries are necessary when we have to calculate aggregates in non-SELECT operations (including reasoning queries); this is because aggregate calculations are only allowed in SELECT – a SELECT will perform the calculation and will provide it to the reasoning rule.

The next example generates the type WellConnectedArtist for all individuals whose number of collaborators is higher than 2.

INSERT **{?person a :WellConnectedArtist**}

WHERE

{

SELECT **?person** (COUNT(DISTINCT ?collaborator) AS ?collabCount)

WHERE

{

?person (:workedWith|^:workedWith) ?collaborator.

}

GROUP BY ?person

**HAVING(?collabCount>2)**

}

In some cases, between the inner query and the main query the passed variables must be involved in some intermediate calculations. For this purpose we use BIND, which is nothing else than a traditional variable assignment.

In the example below we look for all actors and **build a new ID for them, as well as a new type** declaration (we also make them Persons). The new IDs are dynamically built out of the existing actor names through some simple transformations: remove spaces and concatenate the prefix address. To avoid modifying the database, test this as a CONSTRUCT:

CONSTRUCT {**?newID** a :Person}

WHERE

{

{

SELECT ?name

WHERE {?oldID a :Actor; :hasName ?name}

}

**BIND (IRI(CONCAT("http://buchmann.ro#",REPLACE(?name," ",""))) AS ?newID)**

}

Notice how BIND reads the result of the inner quey (?name) and applies some transformations to produce ?newID – this is pushed up in the CONSTRUCT rule body. The transformations can be easier understood if we separate them in successive BINDs:

CONSTRUCT {?newID a :Person}

{

{

SELECT ?name

WHERE {?oldID a :Actor; :hasName ?name}

}

**BIND(REPLACE(?name," ","") AS ?nameWithoutSpace)**

**BIND(CONCAT("http://buchmann.ro#",?nameWithoutSpace) AS ?uriString)**

**BIND(IRI(?uriString) AS ?newID)**

}

* first REPLACE removes all spaces
* then CONCAT concatenates the prefix address
* finally IRI converts the concatenation result (a string) to a URI and stores it in ?newID, to be used by CONSTRUCT

*You may have wondered how did we decide the way IDs should look – in SQL databases, they are often autoincremental numbers but this is not possible here, because they need to act as IDs across the entire Web! In our examples they are more or less similar to the names (labels) but this is not mandatory.*

*The form of URIs must be planned in advance to avoid accidental creation of the same ID for different entities. The prefix address takes care of this on a global level but must also ensure it on a local level – by building the URI according to some internal organization rule (e.g. derive it from the name, but if the name is already taken concatenate some numbering, a UUID code or other differentiator).*

*The CONSTRUCT exemplified here with BIND operations is the typical pattern for building a new URI at the moment when a new "record" must be inserted (however, CONSTRUCT should be replaced with INSERT to perform the actual insertion).*

An alternative to BIND, also used for variable assignment, is VALUES:

* it is more powerful than BIND in the sense that it can assign to multiple variables simultaneously, and each variable can get multiple values
* it is weaker than BIND in the sense that it cannot calculate expressions, it can only assign simple values

There are several use cases when it is beneficial:

* when the same query must be executed repeatedly for a set of URIs – e.g. we only want to get the movies directed by JohnMcT, JamesCameron and McG:

SELECT ?film

WHERE

{

?director :directorOf ?film

**VALUES ?director {:JohnMcT :JamesCameron :McG}**

}

(we can get the same result by applying FILTER IN or UNION, but if the list is long, VALUES becomes more practical and easier to inject in the query at script-level)

* when we want to avoid subqueries by passing data between chained queries on script level (in the programming language); the example below starts with a reasoning that generates the :totalBudget relationship between directors and the cummulative budget of their movies:

CONSTRUCT {?director :totalBudget ?Tbudget}

WHERE

{

**# first the subquery computes the total budgets, to pass it to the reasoning rule**

SELECT ?director (SUM(?budget) AS ?Tbudget)

WHERE

{

?director :directorOf/:hasBudget ?budget.

}

GROUP BY ?director

}

Now let's break this complex operation in two distinct steps: first we execute the subquery

SELECT **?director** (SUM(?budget) AS **?Tbudget**)

WHERE

{

?director :directorOf/:hasBudget ?budget.

}

GROUP BY ?director



Then later, on script level, we pick the results and inject them in the CONSTRUCT rule:

CONSTRUCT {?director :totalBudget ?Tbudget}

WHERE

{

**VALUES (?director ?Tbudget)**

**{**

**(:ShBlack 15000000)**

**(:McG 20000000)**

**(:JamesCameron 24340000)**

**(:JohnMcT 18000000)**

**}**

}

*Moreover, the source of injected values does not have to be the preceding query – VALUES is especially useful if the injected data has non-RDF origin – a front-end form, an API, an SQL query.*

*Example 3. Subqueries for successive aggregate calculations*. The next example lists all persons together with the percentage of people they collaborated with, out of the total number of persons. This cannot be done in a single aggregate calculation:

* first we need to obtain the number of collaborators for every individual
* then we need to obtain the total number of individuals (actors and directors)
* then we compute the fraction

SELECT ?person ?collabCount ?personCount (?collabCount\*100/?personCount AS ?percentage)

WHERE

{

**# first subquery gets all individuals and their number of collaborators**

{

SELECT ?person (COUNT(DISTINCT ?collaborator) AS ?collabCount)

WHERE {?person (:workedWith|^:workedWith) ?collaborator.}

GROUP BY ?person

}

**# second subquery counts all actors and directors once**

{

SELECT (COUNT(DISTINCT ?person) AS ?personCount)

WHERE {?person a ?type . VALUES ?type {:Actor :Director}}

}

}

**# results of inner queries are used by the main query to compute the percentage**

Try to extend this example to compute the percentage out of the collaborators that are reachable along a chain of workedWith relationships for a selected individual.

*Of course, such complex queries can also be avoided by performing the computation on script level, in the programming language from which we send the queries – just like with SQL, this depends if we want the calculation effort to be made by the graph server or by the client app.*

*Example 4. The last use case, where a SELECT must filter its results based on the results of another SELECT query* is covered in the next section (it is an exclusion pattern):

**Queries with exclusions (negation)**

There are several ways of excluding results from a query:

* **avoiding properties:** this is achieved when the modifier ! is applied on some properties; the next example returns all things to which Arnold has a relationship, with the exception of his label, type and birth information:

SELECT ?x

WHERE

{

:Arnold **!**(a|:hasName|:birthInfo) ?x

}

* **negating filters:** this is achieved by applying the traditional logical negation (NOT or !) on a filter condition; the next example will display actor-movie pairs, excluding those involving the Predator and Terminator:

SELECT ?x ?y

WHERE

{

?x :playedIn ?y

FILTER(?y NOT IN (:Predator, :Terminator))

}

...or:

SELECT ?x ?y

WHERE

{

?x :playedIn ?y

FILTER((?y!=:Predator)&&(?y!=:Terminator))

}

* **exclusion of subquery results:** this is what we mentioned earlier, it can be achieved with MINUS or FILTER NOT EXISTS on a subquery; the next example displays all movies where Arnold did NOT play – first we obtain all movies, then we exclude those involving Arnold:

SELECT ?x

WHERE

{

?x a :Movie.

**MINUS** {:Arnold :playedIn ?x}

}

...or:

SELECT ?x

WHERE

{

?x a :Movie.

**FILTER NOT EXISTS** {:Arnold :playedIn ?x}

}

*Beware of possible wrong approaches to the last example:*

*SELECT ?x*

*WHERE*

*{*

*:Arnold !:playedIn ?x*

*}*

*- will return things to which Arnold has ANOTHER RELATIONSHIP than :playedIn (so not movies)*

*SELECT ?x*

*WHERE*

*{*

*?actor :playedIn ?x*

*FILTER (?actor!=:Arnold)*

*}*

*- will return movies where there is at least ANOTHER actor attached (those with Arnold will still be returned, if they have additional actors)*

To complicate things a bit, the next example displays directors who did NOT work on Predator nor Terminator – first we look for all directors, then we exclude those involved with Predator or Terminator:

SELECT ?x

WHERE

{

?x :directorOf ?y.

MINUS {

?x :directorOf ?film.

VALUES ?film {:Predator :Terminator}

}

}

Such exclusions, just like any other subquery, can be used in reasoning rules as well. The next example generates recommendations of social connections – for each individual we recommend the collaborators of collaborators, applying two exclusions:

* the individual for which we generate the recommendation (makes no sense to connect someone with themselves);
* persons that are already direct collaborators:

CONSTRUCT {**?x :getSocialRecommendation ?y**}

WHERE

{

?x (:workedWith|^:workedWith)/(:workedWith|^:workedWith) ?y.

**FILTER (?x!=?y)**

**FILTER NOT EXISTS {?x (:workedWith|^:workedWith) ?y}**

}

*Graph-based recommender systems heavily rely on this kind of reasoning!*

## Lucene searches

Lucene is a search engine library that is commonly used to implement search operations over large textual content present in documents or databases – typically more powerful than what a query based on REGEX string patterns can achieve.

In order to activate the possibility to use Lucene options in SPARQL queries, we have to build indexes that are accessible through **Lucene connectors**:

* + Lucene indexes only individuals (URIs) that have a type declared
  + Lucene searches will return only URIs, but other kinds of nodes will be reachable in the same SPARQL query
  + One connector can search along multiple RDF properties, for multiple entity types, possibly avoiding nodes that fit certain patterns
  + Lucene supports *multi-field search patterns* – this implies that it can report on which property it found results (and also where exactly in the found text value, if the graph stores large text nodes)
  + Lucene synchronizes automatically with every change in the database content

We did not demonstrate the Lucene search until now because there were not entity types available in the graph. We still don't have all types, but we have sufficient to build the first Lucene connector: Actors, Directors and Movies.

There is a distinction between *what is searched/found* and *what is returned/indexed* – they can be the same, but with graphs it's more helpful to return something connected to what was found, not directly the found node. Most of the time

* we search for a string (a text node)
* but the really useful result (that should be returned) is an ID - connected directly or indirectly to the found text (and possibly used in subsequent queries).

The first step is to plan what should be indexed (i.e. returned) and what should be searched.

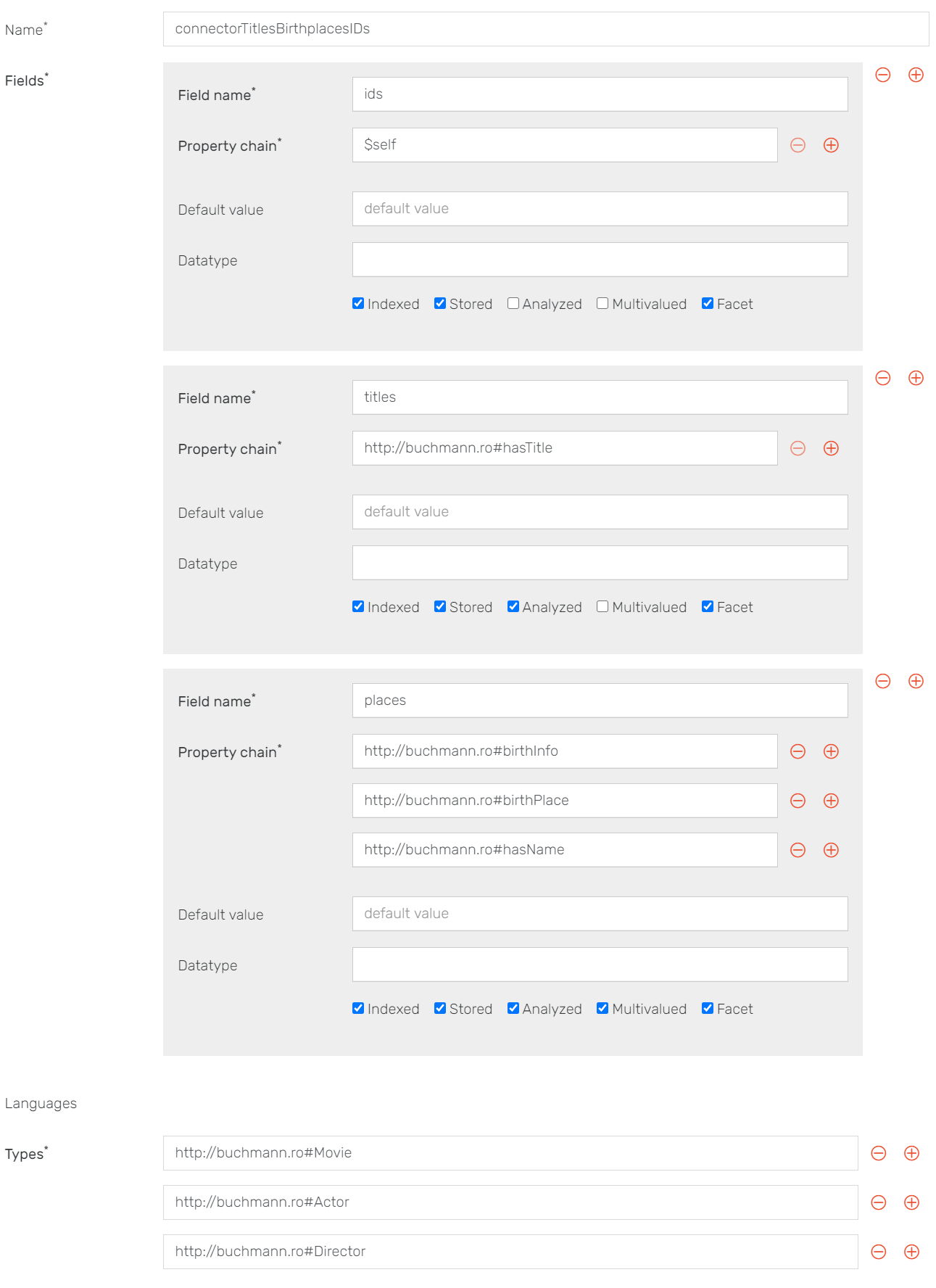
* We want to return actors, directors or movies (their IDs);
* But the search will be applied to titles, birthplaces and IDs (no actor/director names!)

To create a Lucene connector, use the menu Setup-Connectors-New Lucene Connector. All settings can be set in the provided screen:

* **Name**: *connectorTitlesBirthplacesIDs* (this will be invoked in SPARQL queries to trigger the search)
* **Types**: indicate the types of entities to be indexed (returned) – movies, actors, directors
* **Fields**: unlike a simple index, the connector supports *multi-fielded searches*, where a "field" is a shortcut name mapped on a chain of RDF properties. All "fields" together establish how much of the graph becomes searchable:
  + The "titles" field will perform searches in everything reachable through :hasTitle (i.e. movie titles)
  + The "places" field will perform searches in everything reachable through the chain :birthInfo/:birthPlace/:hasName (i.e. birth places labels)
  + The "ids" field will perform searches in all IDs of the types specified above (i.e. limited to movies/actors/directors): exceptionally, when we want to make the indexed URIs themselves searchable we write **$self** instead of a property chain
  + For each field several boxes can be checked/unchecked – as a general rule:
    - we uncheck *Analyzed* for those fields where URIs are searched, or where strings should be treated as a single "word" (unlike the default behaviour of breaking every text value in words and supporting word-level search patterns; i.e. it should also be unchecked if searches such as A\* should look for A at the beginning of the entire value and not at the beginning of a word);
    - we uncheck *Multivalued* for those fields where the property has a single value (e.g. the only multivalued field is the place name, since it has values stored in two languages)
* a **View SPARQL** button at the bottom shows a SPARQL query that can be executed (in the regular SPARQL screen) to build the connector; this is not necessary, but it can be handy to keep your connector settings somewhere in text form and quickly recreate them through queries.

Finally, the **OK** button builds the connector.

For some reason, the current GraphDB version does not allow for modifying an existing connector – notice that on the right side only buttons for deleting, repairing and copying are provided. The practical way of making a minor adjustment is to create a copy, change what needs to be changed and give the connector a new name. Later, remove all unused connectors.



With the connector created, a simple search can be performed with the query:

PREFIX luc: <http://www.ontotext.com/connectors/lucene#>

PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?result WHERE

{

?search a inst:connectorTitlesBirthplacesIDs ;

luc:query "A\*" ;

luc:entities ?result .

}

The query needs:

* to look for an instance of the connector (having as ID the name we gave it)
* to indicate the search pattern (words starting with A)
* to store the returned stuff in a variable (?result)

The results should be Arnold, Avatar, JohnMcT and TimBurton. Maybe it's not clear *where exactly did it find a word starting with A* – we can add some options to get this information:

PREFIX luc: <http://www.ontotext.com/connectors/lucene#>

PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?result **?foundField ?foundText** WHERE

{

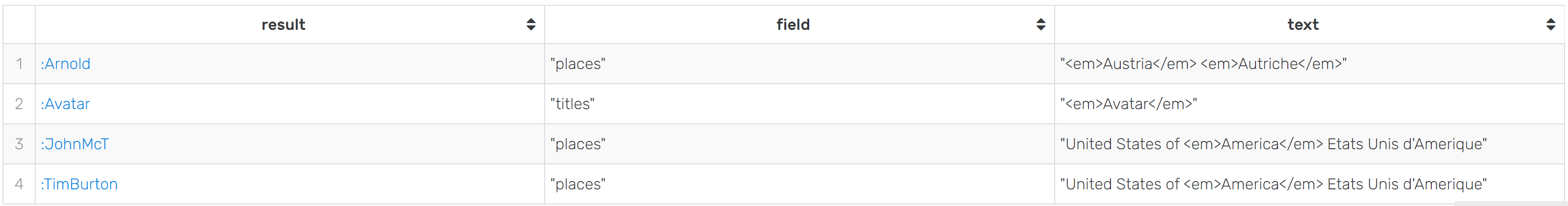
?search a inst:connectorTitlesBirthplacesIDs ;

luc:query "A\*" ;

luc:entities ?result .

**?result luc:snippets [luc:snippetField ?foundField; luc:snippetText ?foundText].**

}



It does not show us exactly the RDF properties, only the Lucene "field" – we should be able to consult in our connector settings to which RDF properties they refer. In the case of snippetText, we see all values merged together where the *Multivalued* setting was used (i.e. country names) and the marker <em> on the exact word that was found (useful if the searched string is a larger text node).

We can see that all results were found in place labels and movie titles – what about searching in the IDs themselves (the "ids" field)? With URIs the behaviour is a bit quirky[[9]](#footnote-9):

* URI searches do not return snippets (so the :snippets section of the query should be marked with OPTIONAL or removed)
* URIs are searched in their full form – i.e. not *Arnold*, but *http://buchmannr#Arnold* (so a search starting with *http\** gives us all the indexed URIs)
* The Lucene treatment of URIs is strange – uppercase letters must be escaped with \\ (but only in URIs)

So a search that looks for *Mc* only in URIs (i.e. the Lucene field "ids") is written as follows:

PREFIX luc: <http://www.ontotext.com/connectors/lucene#>

PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?result WHERE

{

?search a inst:connectorTitlesBirthplacesIDs ;

luc:query **"ids:\*\\Mc\*"** ;

luc:entities ?result .

}

Notice the "ids" particle which indicates the possibility to perform *multi-fielded searches*. If we want to search for *B* either at the beginning of an ID's local part (after #) or at the beginning of a title, the search pattern must be targeted to the desired "Lucene fields" (for ids we apply the mentioned patching):

PREFIX luc: <http://www.ontotext.com/connectors/lucene#>

PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?result WHERE

{

?search a inst:connectorTitlesBirthplacesIDs ;

luc:query **"ids:\*#\\B\* titles:B\*"**;

luc:entities ?result .

}

Lucene provides a form of built-in reporting called "faceted search": we get a full list of all available values under each "field" and how many times it is reachable from indexed nodes[[10]](#footnote-10):

PREFIX luc: <http://www.ontotext.com/connectors/lucene#>

PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?n ?v ?c WHERE

{

?search a inst:connectorTitlesBirthplacesIDs ;

**luc:facetFields "titles,places";**

**luc:facets [luc:facetName ?n; luc:facetValue ?v; luc:facetCount ?c].**

}

The report shows that most titles and place names are uniquely used, except for the United States of America (twice).

One powerful feature of Lucene connectors is the ability to fine-tune exactly what should be findable and what not. Two use cases will be presented as follows:

* Actors/directors should only be findable if they have both birth place and birth date declared; none of the others should be indexed (and neither TimBurton);
* Movies should be findable even if they don't have a title, nor a URI (\_:somemovie lacks both, therefore it is not currently findable).

These issues can be tackled with an **entity filter** – this is a condition written in a dedicated syntax[[11]](#footnote-11) in the Entity filter box of the connector settings (copy the current one, make the adjustments, update the connector name to *connectorConstrained*). The simplest and most commonly used filter looks like this:

bound(?places)

This makes results of the "places" field mandatory – i.e. things will be findable/indexed only if they have birth places (if they only give results for "ids" or "titles" they are not covered by the connector index!)

Now we have to tweak this formula to also allow movies:

bound(?places) || bound(?titles)

Furthermore, for actors/directors, if they have places they should also have birth dates. But this is not covered by the current Lucene "fields". We could add a new "field" just for the sake of this exclusion, but we don't want birth dates to be searchable, we only want to use them as a constraint. This is achieved by a more complex filter:

bound(parent(parent(?places)) -> <http://buchmann.ro#birthDate>) || bound(?titles)

This is an OR filter allowing for two kinds of nodes to be indexed by the connector. The interpretation of this filter is:

* The left side of || says that "places" are findable only if, when you move back two connections in the graph (stated with *parent(parent())*) you find there the birthDate property (stated with *bound()* );
* The right side of || makes sure that "titles" results remain findable;
* However, the "ids" field is not involved – we are not forbidden to use it in the search pattern, and it will still look into URIs, but only for those individuals that are also findable by the filter criteria.

To test this, run the SPARQL query that lists everything that is findable according to the new connector:

PREFIX luc: <http://www.ontotext.com/connectors/lucene#>

PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?result WHERE

{

?search a inst:connectorConstrained ;

luc:query **"\*"**;

luc:entities ?result .

}

The result shows movies and two persons – those having both birth place and birth date. However, not all movies are findable – remember the anonymous \_:somemovie, not having ID, nor a title, which means that it is excluded by bound(?titles).

We can make it findable if we replace the condition of having a title with the condition of being findable by "ids" and having the type Movie. The **$self** property covers not only URIs, but also underscore IDs (anonymous nodes).

bound(parent(parent(?places)) -> <http://buchmann.ro#birthDate>) || ($ids type in (<http://buchmann.ro#Movie>))

Let's push it a bit further: in the "titles" fields there is a Default value box – put in there "notitle". This will be a default string – not stored in the database, but memorized by Lucene as a placeholder for all things that have no results on the :hasTitle property.

Create a new connector with these changes instead of the previous one – name it connectorConstrained2. Check again for all findable things and the anonymous :

PREFIX luc: <http://www.ontotext.com/connectors/lucene#>

PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?result WHERE

{

?search a inst:connectorConstrained2 ;

luc:query **"\*"**;

luc:entities ?result .

}

The anonymous node is now listed. Even more, it is now findable with a search for the placeholder "notitle" – a useful trick to find things that have no labels but are still present in the database and may be useful to a client that performs searches on movies.

PREFIX luc: <http://www.ontotext.com/connectors/lucene#>

PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?result WHERE

{

?search a inst:connectorConstrained2 ;

luc:query **"notitle"**;

luc:entities ?result .

}

There is a minor inconvenience – the two persons still covered by the indexing also got this "notitle" placeholder – it's not possible to limit the "default value" but it's also not a problem, since it is not stored in the database (it's only injected by Lucene in search results). Don't forget that searching queries are still SPARQL queries so it's very easy to make sure that what we get are only Movies (lacking a title but having some other useful information attached):

PREFIX luc: <http://www.ontotext.com/connectors/lucene#>

PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?result ?b WHERE

{

?search a inst:connectorConstrained2 ;

luc:query **"notitle"**;

luc:entities ?result .

**?result a :Movie; hasBudget ?b**

}

1. Notice however that, while adding an indirect connection, we did not remove the direct connection (that would require a DELETE operation) [↑](#footnote-ref-1)
2. The NetworkX library in Python provides some powerful tools for this purpose [↑](#footnote-ref-2)
3. Those of you who recognize such multipliers will guess that we can also use \* (zero or many, optional and repeatable) or ? (zero or one, optional but not repeatable) [↑](#footnote-ref-3)
4. See https://www.stardog.com/docs/#\_path\_queries\_2 [↑](#footnote-ref-4)
5. See https://www.w3.org/community/sparql-12/ [↑](#footnote-ref-5)
6. See https://networkx.github.io/ [↑](#footnote-ref-6)
7. See the full list at <https://www.w3.org/TR/sparql11-query/#setFunctions> [↑](#footnote-ref-7)
8. We would need to decide how to treat movies that don't have a budget, but we simplify and focus on the aggregate example [↑](#footnote-ref-8)
9. Perhaps a future patch cleans these aspects [↑](#footnote-ref-9)
10. For reasons probably related to the aforementioned patching, nothing is reported for fields where URIs are searched [↑](#footnote-ref-10)
11. See the expression syntax at https://graphdb.ontotext.com/documentation/standard/lucene-graphdb-connector.html#advanced-filtering-and-fine-tuning [↑](#footnote-ref-11)