Semantic Technology:

Installation of GraphDB +

Basic graph queries +  
the RDF-star extension

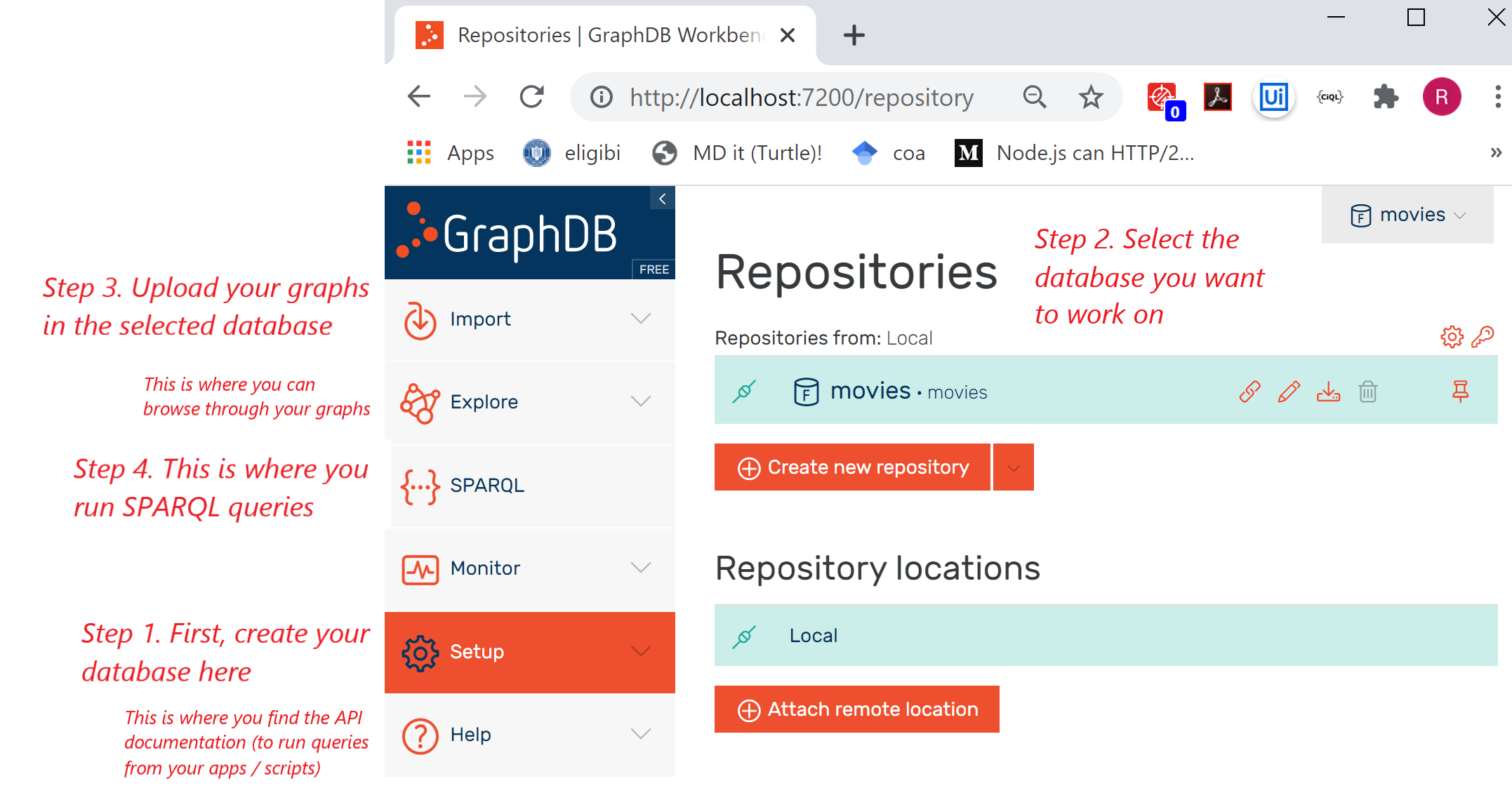
**Objectives of this tutorial:**

* To **create your first RDF graph database** on GraphDB (from a given dataset about movies)
* To run your first **graph queries** (SELECT, ASK, DESCRIBE) using the SPARQL language
* To learn the basics of the **RDF-star** syntactic extension

**Installation steps (Windows)**

1. Obtain **GraphDB Free** from http://graphdb.ontotext.com/
2. After starting, its management console will start in browser at localhost:7200

The figure below shows you where you can find the main options in the management console, as well as the main steps you need to follow in order to exercise the queries of this tutorial.



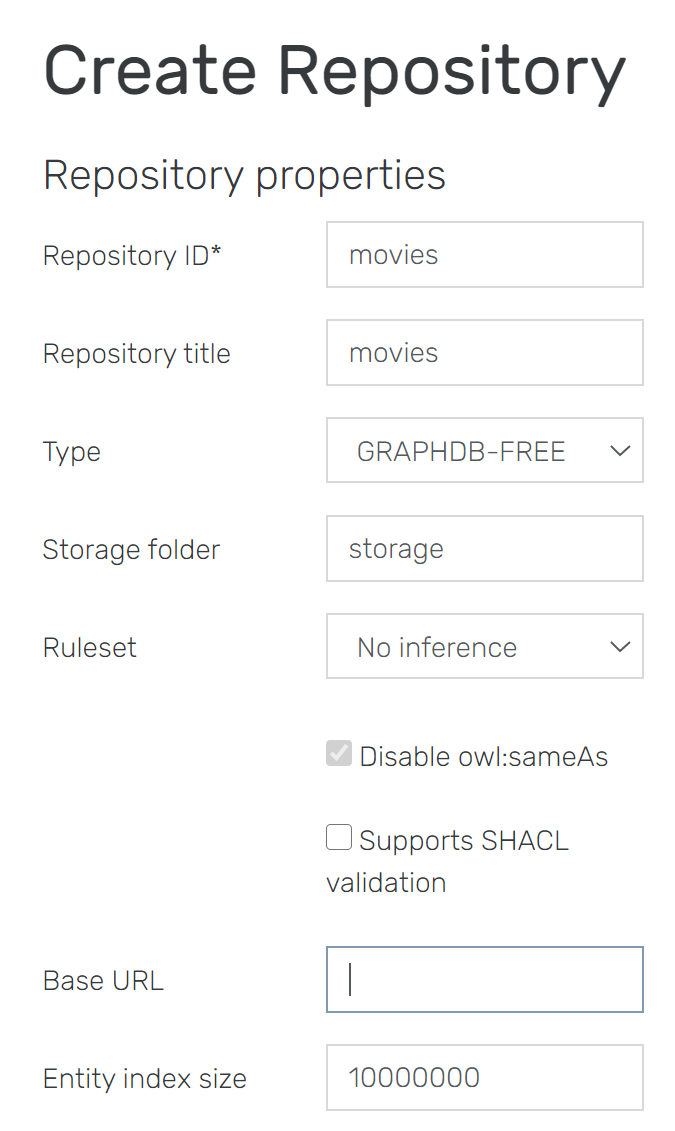
**Create your first graph database**

Use the ***Setup*** menu, and ***Create new repository***. First it will ask yout about the type of repository:

* GraphDB Free (the "normal" one, choose this)
* OnTop (a virtual repository over existing SQL databases, it will allow you to query MySQL with SPARQL without converting tables to graphs)
* FedX (also a virtual repository acting as a gateway for multiple RDF APIs available on the Web, it will distribute federated queries among multiple SPARQL endpoints)

After choosing the normal repository, interact with the following fields:

* *Repository ID and title* – they can be the same (movies)
* *Ruleset* – choose *No inference* to deactivate axiom-based Machine Reasoning; this will improve performance significantly, while dropping certain reasoning features (we will still be able to define some **reasoning rules with the help of the SPARQL query language**!)
* *SHACL validation* – uncheck it to deactivate schema validation (our example **will have no schema** and no data validation possibilities, we focus on queries)



## Create your first RDF graph database

**The content of our first RDF database, first in natural language**

We'll start with an overview of the information we plan to store:

* some information about various relationships between people and things (movies, birth places, movie roles etc.):

James Cameron is the director of Avatar and Terminator.

John McTiernan is the director of Predator and Die Hard and he was born in USA.

Joseph McGinty is the director of a movie (unknown yet) where Sam Worthington played.

Sam Worthington also played in in Avatar where he had the role of Jake Sully.

Arnold Schwarzenegger played in Terminator and Predator; also, he was a governor of California and he was born in Austria.

Linda Hamilton played the role of Sarah Connor in Terminator.

Shane Black played in Predator and is the director of Kiss Kiss Bang Bang.

Tim Burton is the director of Batman and was born in USA.

Jack Nicholson played in Batman.

* some information that actually contains data values (integers, dates):

Terminator had a budget of 64000000.

Predator had a budget of 18000000.

Kiss Kiss Bang Bang had a budget of 150000000.

The unknown movie directed by McGinty, where Sam Worthington played, had a budget of 200000000.

John McTiernan's birth date is 8.01.1951.

Arnold Schwarzenegger's birth date is 10.07.1947.

We have to decide what should be the shape of IDs (URIs).

* We have the choice of adopting IDs from a public source (e.g. DBPedia), because the individuals and things mentioned here are notable enough to have Wikipedia pages (therefore they also have DBPedia identifiers). For example, the DBPedia identifier of John McTiernan is **dbr:John\_McTiernan**[[1]](#footnote-1), for Terminator it is **dbr:The\_Terminator** etc.
* Or, we can improvise our own terms and IDs, attached to a default prefix ( : ) associated with our fictive domain address e.g. http://buchmann.ro#

We'll take the second option (consider the first option as a homework, try to "translate" the version that we're building here to DBPedia terms, including properties available there).

Generally, identifiers will be derived from the names, **by attaching the prefix and removing the space** (spaces are separators between the terms in a statement). We'll make however some exceptions:

* In some cases, the ID will be only a fragment of the full name (Arnold, SamWorth, LindaH, ShBlack) or an abbreviation (KKBB for the movie Kiss Kiss Bang Bang)
* In some cases the ID will be derived from the nickname of a person (McG for Joseph McGinty)
* For locations, the names will be stored in two languages (English and French) and the ID will be based on the English name

*This is just to show that the ID and the name (label) are different things –* ***we should attach a name to every ID****, but they do not have to be similar. Reminder:*

* *Names/labels are to be displayed in a front-end, possibly multilingual (and two different things may have the same name)*
* *IDs act as global identifiers (no two things in the entire Web should be identified with the same full URI!)*

Now let's translate the content to an RDF graph.

**The graph, written in TriG syntax**

Reminder: TriG is almost the same as Turtle. It only adds the possibility to use accolades as delimiters of the graph content, and to add a graph identifier in front. This makes it possible to have multiple graphs in the same text file.

Here is the full dataset, written in a graph identified by :mymoviegraph. Some annotated comments should help you remember the main syntactic rules*:*

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

The entire content is included in a **graph with this ID**

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

:mymoviegraph

{:JamesCameron :directorOf :Avatar,:Terminator;

Here we have **multiple statements about the same individual**, therefore we group them with , and ; as separators

:hasName "James Cameron".

:JohnMcT :directorOf :Predator, :DieHard;

:hasName "John McTiernan";

:birthInfo **\_:birthdetails.**

**Anonymous nodes as data structures**, to keep together the birth date and birth place.

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

**:birthPlace :SUA.**

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

**Anonymous node as placeholder** for some unknown movie where different persons were involved

:hasName "Joseph McGinty";

:hasNickname "McG".

:SamWorth :hasName "Sam Worthington";

:playedIn **\_:somemovie**;

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

**Anonymous nodes for a complex relationship** with 3 participants (actor, movie, role)

**: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 .

Some parsers will think that a number followed by a dot is a decimal number. To avoid this confusion, it's a good habit to leave a space before the final dot.

:Avatar :hasTitle "Avatar";

:hasBudget 237000000 .

:DieHard :hasTitle "Die Hard".

:Predator :hasTitle "Predator";

:hasBudget 18000000 .

**Every ID has a label attached** (more or less similar)

:KKBB :hasTitle "Kiss Kiss Bang Bang";

:hasBudget 15000000 .

\_:somemovie :hasBudget 200000000 .

:JakeSully :hasName "Jake Sully".

:SarahConnor :hasName "Sarah Connor".

**Labels** written in multiple languages

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

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

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

}

A visualization of the graph can be generated with the following tools:

<http://www.ldf.fi/service/rdf-grapher>

https://issemantic.net/rdf-visualizer



**Study the graph a bit, understand its structure**

Notice the following aspects:

1. It is a **schemaless graph** – we don't have entity types, we did not specify WHAT are the things mentioned here. That would be expressed in standard statements such as

:Arnold a :Actor.

We intentionally neglected this to show later how entity types can be generated automatically by Machine Reasoning.

2. **The use of properties is somewhat irregular**:

* The actor-movie relationship is expressed in two different ways (*playedIn* or *playedTheRole* followed by an anonymous node to also connect the role)
* The birth place is declared in two different ways (for TimBurton it is directly attached, for the others it is grouped together with the birth date, by an anonymous node)
* The entities do not share a common schema, we don't know the same information about all of them (some movies have a budget, others don't etc.)

These are not errors, they are inconveniences – if we know they are present, we can prepare queries that will find all the needed combinations. However, it's a good idea to remove irregularities in order to make the graph easier to navigate by others. This cleaning up will also be performed later.

3. It demonstrates **three use cases for anonymous nodes** (see also the previous tutorial):

* The complex (non-binary) relationship (between actor, character, movie):

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

**:inMovie :Terminator]**.

* The "data structure" (birth date and birth place grouped together):

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

**:birthPlace :Austria].**

* The "placeholder" (the same "unknown" movie mentioned in different places, forcing us to give it an underscore ID instead of square brackets):

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

......................

:SamWorth :playedIn **\_:somemovie**;

......................

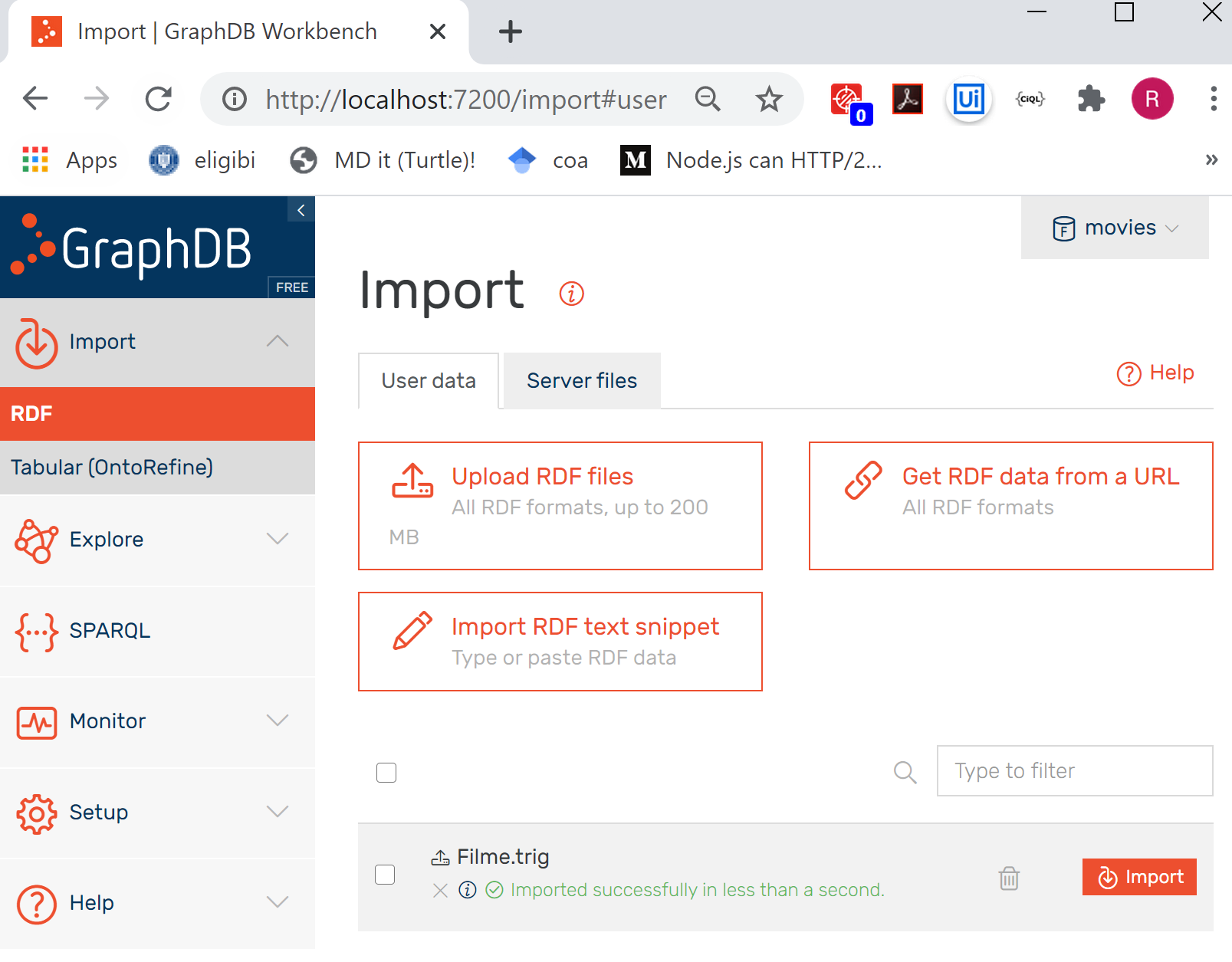
**\_:somemovie** :hasBudget 200000000 .

*Try to locate the anonymous nodes in the graph visualization at https://www.ldf.fi/– see the green circles.*

**Upload the graph in GraphDB**

In the **Import** menu, use *RDF* to upload graphs written in various RDF syntaxes (Turtle, TriG, N-triples etc.)

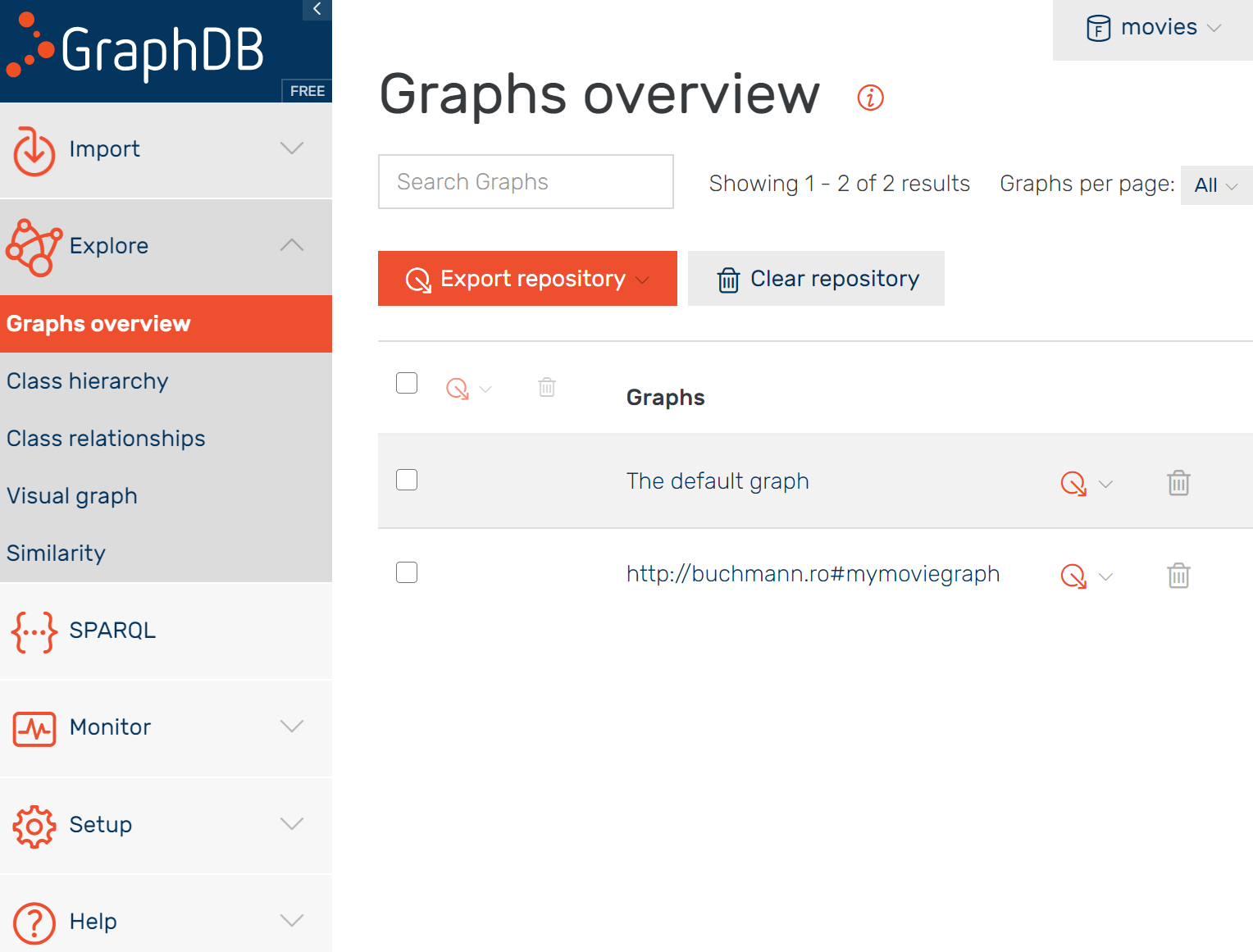
In this case, we have the graph written in a TriG file – either upload it as a file (*Upload RDF Files*) or copy-paste it in *Import RDF text snippet*. Don't forget to press the Import button, so that you can see the successful Import confirmation (you might get an additional pop-up window, *Import settings*, where the import must be reconfirmed).



**Browse the graph content**

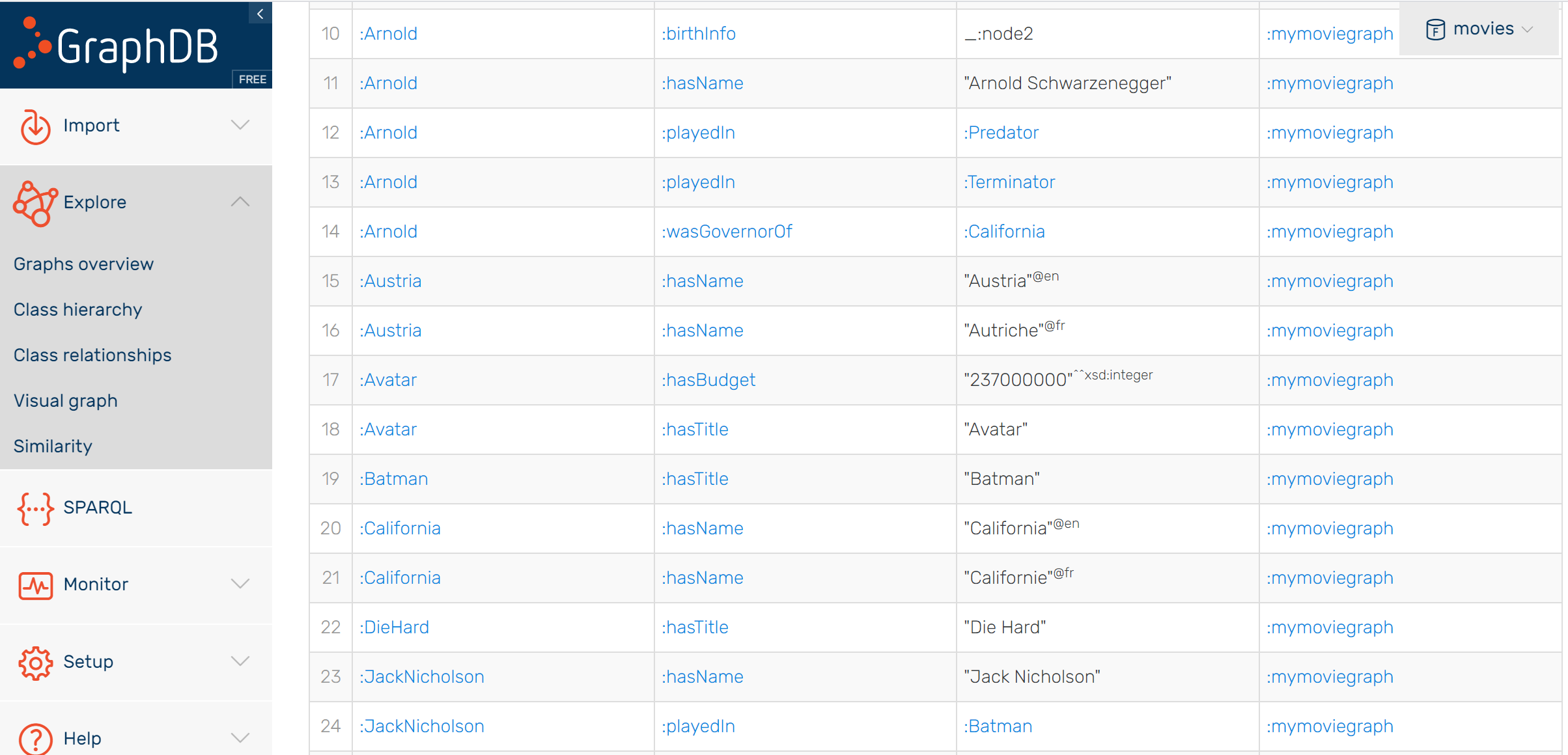
In the **Explore** menu, with *Graphs overview* you can browse and manage your graphs:

* You get a list of all graphs (multiple graphs can be created in the same database); the *default graph* will contain any data that was uploaded without designated a targeted graph ID
* You can notice buttons for deleting a graph, or for exporting them in various syntaxes
* And of course a button to quickly empty the entire database

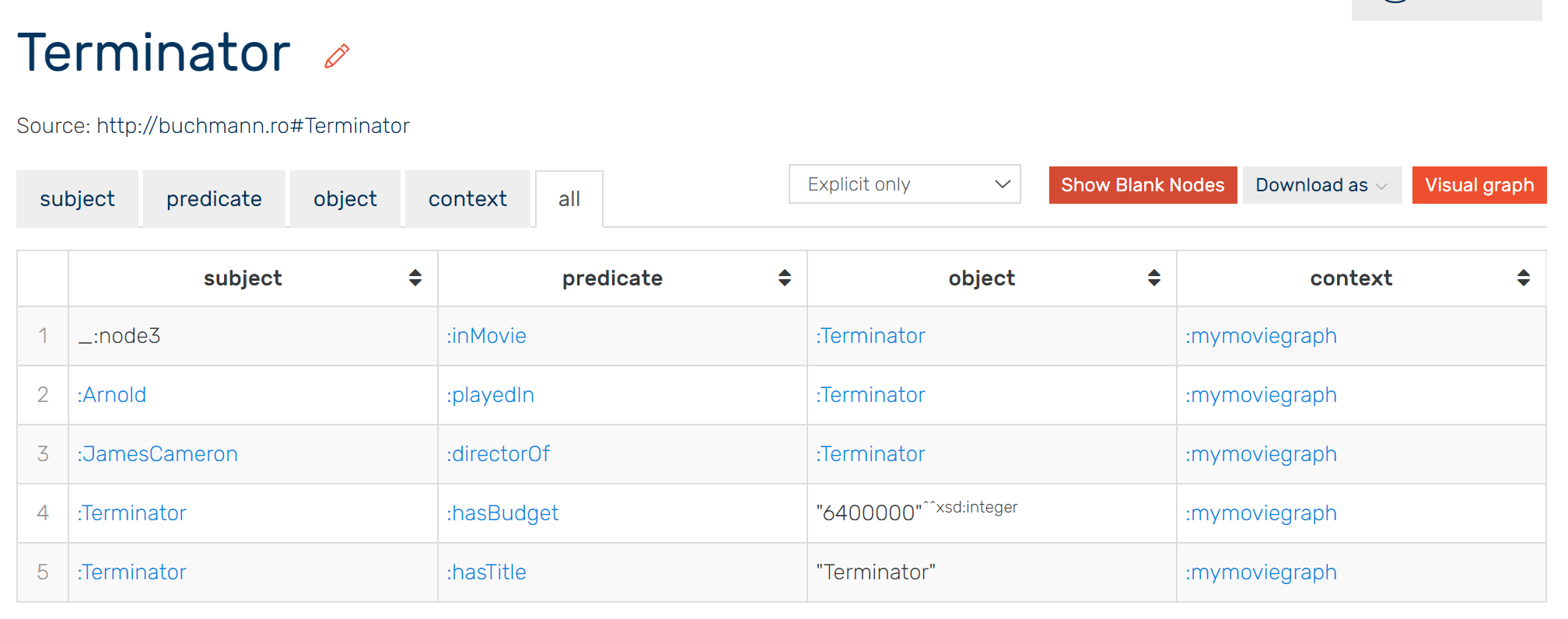


Click on mymoviegraph to browse its contents in a tabular form

* notice that the anonymous nodes changed their IDs – we mentioned that it does not matter how we call them
* also notice that all URIs are clickable, whereas the data values are not

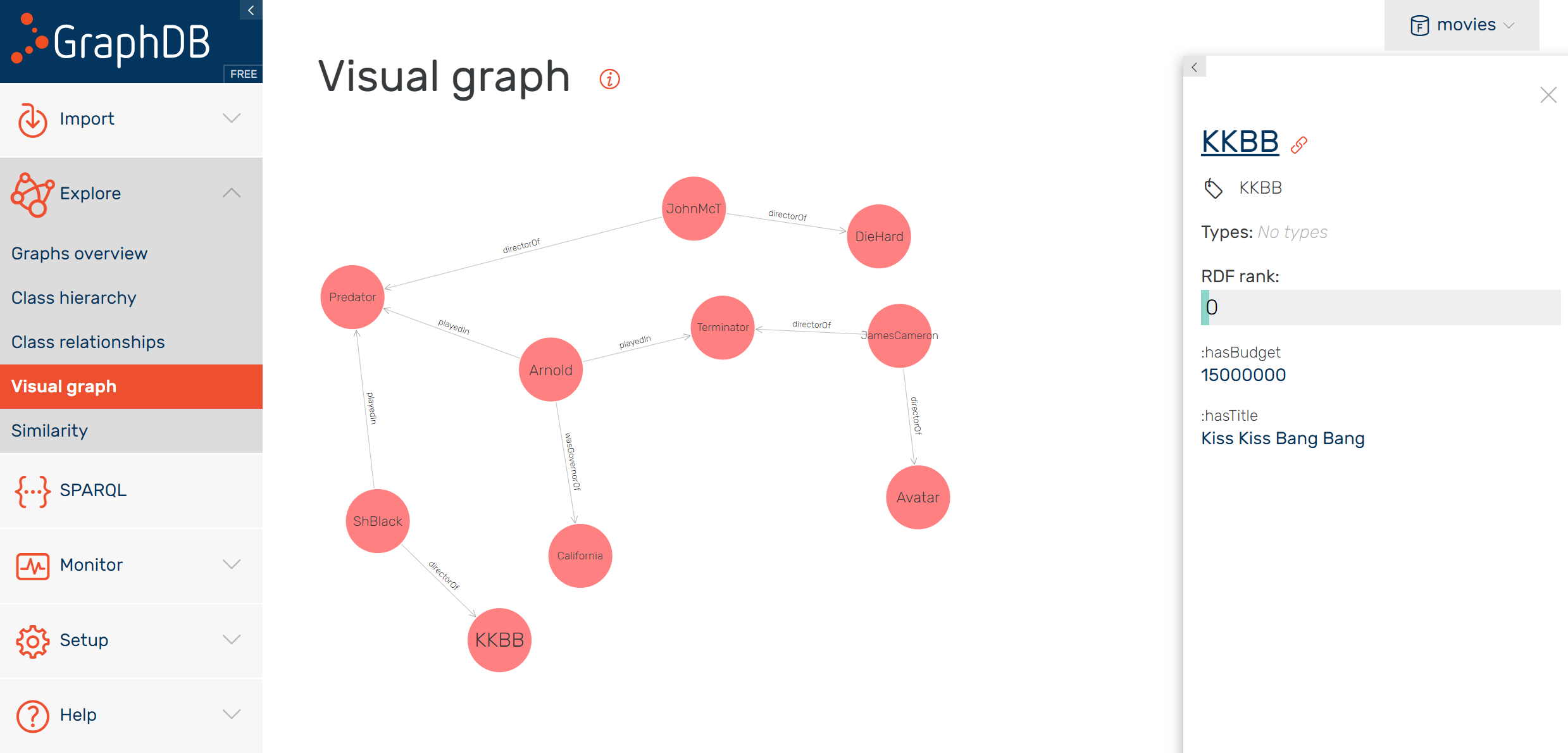


Click on *Terminator* to see strictly the subgraph connected to that node (various options above the table allow you to filter only statements where it appears as a subject, property, object; or, to see *all* of them):



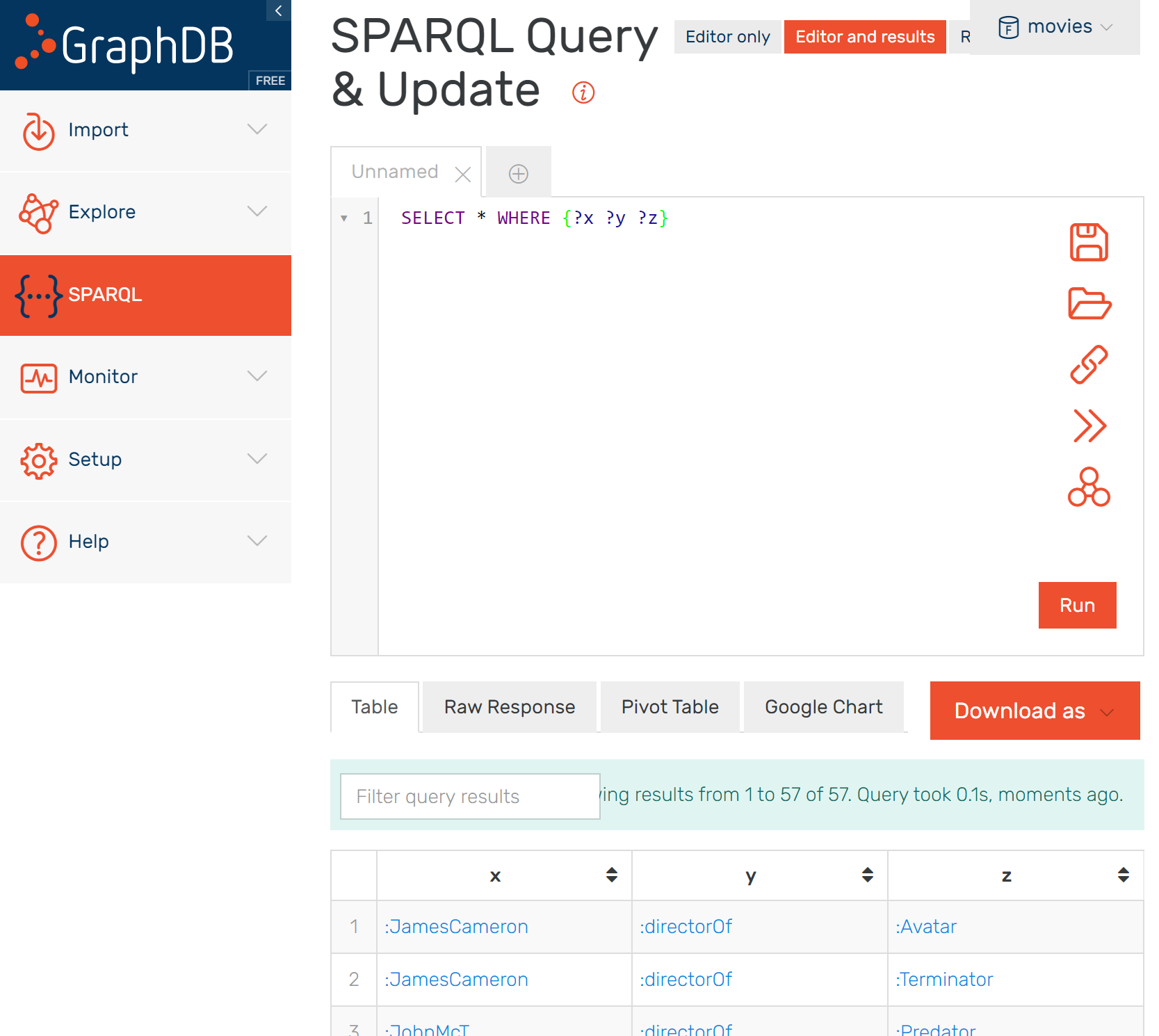
Also in the **Explore** menu, a *Visual graph* option allows you to interact with the graph visually. It does not display a full visualization, but instead:

* It asks you to type the **node ID from which you want to start** (e.g. type *:Arnold* in the box *Search RDF resources*); if you cannot see search suggestions in the searchbox, go to Setup-Autocomplete to generate an autocompleter index for all search boxes in GraphDB
* Then click by click you can expand the graph starting from that node. There is an important limitation in the free version of GraphDB: the expansion stops at anonymous nodes[[2]](#footnote-2)
* The data values are also not shown in the visual graph; however, they are **displayed in the right side panel**, whenever you select an ID (data would overload the visualization)



## Finally, some graph queries

Write all your queries in the **SPARQL** screen, where a query editor is provided. See below the simplest query, which displays the entire content of the database – i.e. all statements, regardless of what is the subject (x), property (y) or object (z):



Query results are displayed in the table below – i.e. if you don't switch off the button **Editor and Results** visible above. Results can be downloaded in various formats, or visualized as Charts (if that makes sense, i.e. if we obtain quantitative data that can be plotted on a type of Google Chart).

Here is a list of queries to try:

* Display all subjects of all statements

SELECT ?x WHERE {?x ?y ?z}

* The same, but display each subject only once

SELECT **DISTINCT ?x** WHERE {?x ?y ?z}

* Display all statements ordered by property

SELECT \* WHERE {?x ?y ?z} **ORDER BY ?y**

* Display the first 10 statements ordered by subject ID

SELECT \* WHERE {?x ?y ?z} **ORDER BY ?x LIMIT 10**

* Display all statements that contain data values, regardless of datatype

SELECT \* WHERE {?x ?y ?z. **FILTER ISLITERAL(?z)**}

* Display all statements that contain numeric data values[[3]](#footnote-3):

SELECT \* WHERE {?x ?y ?z. **FILTER ISNUMERIC(?z)**}

All of these are simple generic queries, not very useful in general. SPARQL queries should look for some specific IDs or statement patterns to obtain useful information. For example, in which movies did Arnold play[[4]](#footnote-4)?

**PREFIX : <http://buchmann.ro#>**

SELECT ?x

WHERE

{

:Arnold :playedIn ?x

}

Notice that prefix declarations MUST be included for those prefixes used in the query! This can be annoying, but GraphDB will automatically insert them when we type the character ":" (however, if you copy-paste queries instead of typing them, this will not happen, so make sure that the prefix declaration is not removed – we will not replicate it for every query in this tutorial!)

## More sophisticated SELECT queries

**Two syntactic styles for queries: "paths" or "Turtle-style" queries**

Notice that in the previous query all results are URIs (IDs), since those are the nodes directly connected to :Arnold (which is also an ID). Query results to be displayed in a front-end should not be IDs – our queries must look for names/labels, in this case the movie titles:

SELECT ?x

WHERE

{

:Arnold :playedIn**/:hasTitle** ?x

}

Or, we can navigate the path from Arnold's full name to his movies' titles (useful when IDs are less reader-friendly and we want to avoid typing them, e.g. when IDs have been generated from some existing primary keys):

SELECT ?x

WHERE

{

"Arnold Schwarzenegger" **^:hasName**/:playedIn**/:hasTitle** ?x

}

The character ^ indicates that the edge *:hasName* must be navigated in reverse (here, from label towards ID), therefore from the statement object to the statement subject.

SPARQL queries can be written in two syntactic styles:

1. **As "property paths"**, like this example – we use slash characters to build a path "hop by hop" and the character ^ to indicate when one step must be navigated in reverse;
2. **By mirroring the original syntax** (same principle as in GraphQL) – the query will use "Turtle patterns", i.e. the Turtle syntax with some variables in it (placeholders or the elements to be returned)

See below the same query rewritten in Turtle style:

SELECT ?title

WHERE

{

?idActor :hasName "Arnold Schwarzenegger";

:playedIn ?idFilm.

?idFilm :hasTitle ?title.

}

The query will store the result in ?title. The other two variables (?idActor, ?idFilm) are placeholders to connect the graph pattern from "Arnold Schwarzenegger" to the desired result.

Although the "path" syntax looks simpler, the Turtle-style syntax has two big advantages:

* It is easier to debug – we can test different parts of the Turtle pattern, until we figure out were a mistake was made;
* It can return any intermediate node or edge – see below an example where, for Shane Black, we get not only the movies where he was involved, but also the relationship with those movies (sometimes he's an actor, sometimes he's a director):

SELECT ?relationship ?title

WHERE

{

?idPerson :hasName "Shane Black";

**?relationship** ?idFilm.

?idFilm :hasTitle ?title.

}

This example also reveals the possibility to discover relationships that the query client does not know in advance! Comparatively, the "path" syntax is limited in this sense:

* It assumes that we know the full path to be navigated;
* It can only return the start and end node of the path, no intermediate nodes, no intermediate edges. In other words, the current SPARQL version does not allow us to discover a relationship included in a path:

SELECT ?relationship ?title

WHERE

{

"Shane Black" ^:hasName/**?relationship/**:hasTitle ?title

}

The two syntactic styles can be combined – we can have paths mixed in Turtle patterns:

SELECT ?title

WHERE

{

?id :hasName "Arnold Schwarzenegger";

:playedIn/:hasTitle ?title.

}

Or, we can break a path to pick some intermediate node or edge along the way:

SELECT **?directorName** ?title

WHERE

{

:Arnold :playedIn/^:directorOf/:hasName **?directorName**.

**?directorName** ^:hasName/:directorOf/:hasTitle ?title

}

This example looks for *all movies (titles) directed by those who directed the movies where Arnold played*. But along the way from Arnold to those titles, it "stops" to also pick the names of those directors.

The paths in this query can also be written in reverse order – we start from what we look for (titles) and we end the path at the known information (Arnold):

SELECT **?directorName** ?title

WHERE

{

**?title** ^:hasTitle/^:directorOf/:hasName **?directorName.**

**?directorName** ^:hasName/:directorOf/^:playedIn :Arnold.

}

The navigation direction is of course reversed in this case – therefore we invert the use of ^ (edges that were navigated in reverse are now navigated normally and viceversa).

**Queries in irregular graphs**

By "irregular" we mean that not all nodes comply to a common schema – we know different properties about different things or the same properties are expressed in different ways, e.g.

* not all movies have a budget,
* the birth place is sometimes connected to the person, other times to an anonymous node
* not all actors are connected to their movies with the same relationship

Let's say we want to display the titles together with the movie budgets:

SELECT ?title ?budget

WHERE

{

?film :hasTitle ?title;

:hasBudget ?budget.

}

We notice that the results do not include movies for which we don't know the budget – the statement patterns are connected by something similar to an SQL JOIN, where we only get movies for which both title AND budget are available.

In order to indicate that some property is potentially missing, we mark it with the OPTIONAL clause:

SELECT ?title ?budget

WHERE

{

**?film** :hasTitle ?title.

**OPTIONAL {?film :hasBudget ?budget}**

}

With potentially missing properties, it's often necessary to return a default value instead of null values. The default value can be injected in the query response in multiple ways:

SELECT ?title **(COALESCE(?b,"not available") AS ?budget)**

WHERE

{

?film :hasTitle ?title.

OPTIONAL {?film :hasBudget ?b}

}

This is an example of a query that returns not only variables found in the query patterns, but also a calculated expression. COALESCE takes a list of values and variables and returns the first non-null variable in the list. It is a more convenient version of an IF function that returns the default value when a variable has no value:

IF(BOUND(?b),?b,"not available") AS ?budget

*Obviously, COALESCE and IF are not equivalent: COALESCE can check a longer list of variables (will return the first non-null value), IF can verify more complicated conditions than BOUND.*

We also have an anonymous movie (\_:somemovie) for which we know the budget without knowing the title! This means that we also have to mark the title as potentially missing.

Now we have a problem – neither the title, nor the budget are sufficient to know that something is a movie. We can however rely on the director, because this was declared for all movies:

SELECT ?directorName ?title ?budget

WHERE

{

?idDirector :directorOf ?film; :hasName ?directorName

**OPTIONAL {?film :hasTitle ?title}**

**OPTIONAL {?film :hasBudget ?budget}**

}

Of course, relying on the director is not reliable in the general case – we did this trick because we knew what information is available in the graph. In the general case, to make sure that we collect all movies regardless of what properties they might be missing, the graph should include entity types such as

:Terminator a :Movie.

Currently such type declarations are missing, we will add them in future exercises.

Now let's try to find the actors and the movies where they played – we have a problem because

* some actors are connected directly to their movie (:playedIn)
* others are connected through an anonymous node (:playedTheRole/:inMovie)

Consequently we need something like an OR between different patterns:

1. if we prefer the Turtle-like syntax, this can be achieved with UNION between the alternative patterns:

SELECT ?actorName ?title

WHERE

{

?idActor :hasName ?actorName. # first step: picks all names and IDs

**{?idActor :playedIn ?idFilm}**

**UNION** #step 2: keeps only IDs connected through one of those 2 patterns

**{?idActor :playedTheRole [:inMovie ?idFilm]}**

?idFilm :hasTitle ?title. #last step: from movie IDs if goes to pick movie titles

}

1. if we prefer the path syntax, we need the character | and parentheses to correctly apply the logical operator:

SELECT ?actorName ?title

WHERE

{

?actorName ^:hasName/**((:playedIn)|(:playedTheRole/:inMovie))**/:hasTitle ?title.

}

Another irregularity is the birth place property – for Tim Burton it is directly attached, whereas for the others it is attached through an anonymous node, further connected through :birthInfo to the person. The potential absence of :birthInfo can be indicated with ?

SELECT ?persoana ?loc

WHERE

{

?persoana **:birthInfo?**/:birthPlace ?loc.

}

We can also do this with UNION:

SELECT ?persoana ?loc

WHERE

{

{?persoana :birthInfo [:birthPlace ?loc]}

UNION

{?persoana :birthPlace ?loc}

}

One minor inconvenience is that both variants will return the intermediate anonymous nodes, because those nodes also fit the Tim Burton pattern, i.e. of having a :birthPlace property! We want to skip anonymous nodes by applying a filter that ensures only URIs will be returned:

SELECT ?persoana ?loc

WHERE

{

?persoana :birthInfo?/:birthPlace ?loc.

**FILTER (isIRI(?persoana))**

}

The function isIRI() keeps only URIs. Similar functions exist for testing if a result is anonymous (isBlank()) or if it's a data value (isLiteral()).

*Such irregularities will not typically occur if a graph is created by an application providing traditional CRUD operations. However, they can occur if the graph database accumulates information from multiple sources, each following a different schema or graph design.*

*The graph can be "cleaned up" with the help of Machine Reasoning, as we will see later.*

**String-based queries**

We typically need to return string labels in order to display them in the front-end (the end-user should not perceive if the database is a graph, or SQL, or Mongo etc.)

This comes with a frequent need to filter results by string patterns, or to build strings out of other query results.

The strings we have available in our graph are titles, names of people or countries. Just like in any language, we can use:

* string functions
* regular expressions
* Lucene-based text searches[[5]](#footnote-5) which we'll cover in a separate section since they require some extra configuration effort

Let's query the names of directors and their movies' titles, but keeping only those movies whose titles end in "or":

* The variant using string functions – STRENDS() allows us to test how a string ends[[6]](#footnote-6):

SELECT ?directorName ?title

WHERE

{

?directorName ^:hasName/:directorOf/:hasTitle ?title

FILTER **STRENDS(?title,"or")**

}

* The variant using regular expressions:

SELECT ?directorName ?title

WHERE

{

?directorName ^:hasName/:directorOf/:hasTitle ?title

FILTER **REGEX(?title,"or$","i")**

}

With REGEX(), the argument "i" represents the case-insensitive option, whereas the $ code marks the end of a string (therefore the "or$" expression searches the "or" particle at the end of titles).

The REGEX() function is also applicable to URIs, if we convert them to strings using STR. A typical check on URIs is if they belong to a certain provenance address. The next example returns director-movie pairs but only for those movies that have a URI in the "buchmann.ro" namespace (i.e. \_:somemovie will be skipped):

SELECT ?idDirector ?idFilm

WHERE

{

?idDirector :directorOf ?idFilm

FILTER **REGEX(STR(?idFilm),"buchmann.ro")**

}

Sometimes we want to build front-end messages out of query results – the CONCAT() function can be used to build a string expression in combination with string functions that extract whatever we need to concatenate. The next example extracts with STRAFTER() the directors' family names and builds messages such as "Burton directed the movie Batman":

SELECT **(CONCAT(STRAFTER(?nameDirector," ")," directed the movie ",?title) AS ?message)**

WHERE

{

?nameDirector ^:hasName/:directorOf/:hasTitle ?title

}

String labels may have language codes – e.g. the country names are stored in two languages. We can filter strings by language code with the LANG() function, e.g. to obtain only the French versions of birth places:

SELECT ?name

WHERE

{

?x :birthPlace/:hasName ?name

FILTER **(LANG(?name)="fr")**

}

## Exploring unknown graphs

When we don't know in advance the schema/terminology used by a graph (and we don't have any documentation at hand), we can start "rummaging around" – i.e., exploring the graph contents in a step by step manner:

The following DESCRIBE returns all information available about :Arnold – i.e., all statements that contain the ID of Arnold, including those reachable from Arnold through an anonymous node (e.g. his birth place and birth date):

DESCRIBE :Arnold

We can add a WHERE clause to collect information about more than one single thing. The next example returns all information available about those to which Arnold has a relationship:

DESCRIBE ?x

WHERE

{:Arnold ?relationship ?x}

Some exploration can also be achieved with SELECT – e.g. the next example lists all relationships of Arnold without knowing them in advance:

SELECT ?relatie ?x

WHERE

{:Arnold ?relatie ?x}

There is however a major difference between SELECT and DESCRIBE – the result format. This has major implications for programming RDF-based applications – i.e., we have to know what result format we can expect, to know how to parse it in the preferred programming environment:

* **SELECT returns an array** (can be visualized as a table with one column for each returned variable).
  + This implies that, in a programming language, SELECT results can be iterated (e.g. For Each) and easily displayed as a table in front-end;
* **DESCRIBE returns a set of complete statements, which form a subgraph** (it may look like a table with 3 columns, but it's actually a set of RDF statements).
  + This implies that in a programming language DESCRIBE results should be parsed as a graph object, something that requires a dedicated RDF library (which in turn can provide methods to further extract a convenient array)
  + This also implies that servers can implement the "Dereferencing principle" (the ability to return all URIs connected to one that was accessed as a URL), with a simple script that runs a DESCRIBE query!
  + Finally, it also implies that DESCRIBE is useful whenever we want to manage information as a graph, and not as a traditional structure (dictionary, array etc.). We point to two major use cases for this: **graph analytics** (e.g. using Python's NetworkX library to analyze a social network) and **graph publishing** (e.g. publishing data graphs in JSON-LD format, publishing graphs on an API).

Another exploring query is the one that checks **WHAT ARE the things mentioned in the graph** – however, this is not usable for our current example since we don't have entity types attached to the URI nodes. If we had type declarations, a query to obtain all types of things mentioned in the graph would look like this:

SELECT ?type

WHERE

{ ?x a ?type}

(after we know the types, we can get all things of a certain type, then all relationships of those things etc.)

Another exploring query is the one to check properties that we're not sure if they are available (or are potentially missing for some individuals of a certain type):

SELECT ?type

WHERE

{

?x a ?type

OPTIONAL {?x :property ?y}

}

Finally, we also include in the category of "exploring queries" the ASK operation – it returns a boolean value. Example: *are there any statements about Arnold available in the graph?*

ASK {:Arnold ?property ?y}

This kind of query is useful when some front-end must create new IDs – in that case we want to check, prior to an INSERT, whether the ID to be inserted was already used or not (plus, we concatenate our own prefix to make sure that it belongs to our namespace and other servers would not create it by coincidence).

ASK can be used with complex patterns not only for ID checking, e.g. – *is it true that Arnold played in some movie directed by James Cameron?*

ASK {:Arnold :playedIn/^:directorOf :JamesCameron }

The main advantage of ASK queries is performance – it is sufficient to find a result in order to confirm its existence; and sometimes a confirmation is all we need (e.g. when checking if an ID was already taken). Later we can use DESCRIBE or SELECT to retrieve concrete information about that thing – however if we "attack" the graph database directly with a DESCRIBE or SELECT there's a risk that we will get a very large result set for which the server times out (especially if it's a public SPARQL service).

Of course, RDF servers also have the responsibility of documenting their content – thus allowing clients to learn how to build efficient queries without spending much time with "exploring queries".

## A syntactic extension: RDF-Star

**Why RDF-Star?**

RDF-Star is a non-standard extension of RDF (of both Turtle and SPARQL) – it is not part of the standard, but it will be in the near future, since it already has wide support as an experimental feature in RDF servers.

It was introduced to allow "statements about statements" in a way that spares us of using anonymous nodes, while providing better query performance for such situations. The key use cases for this are:

* **To allow data to be attached to distinct property occurrences, not only to nodes** (e.g. statements that are only true in a limited frame of time/space, statements that have a probability of truth); example: *Patrik lives in Vienna since 2005*.
* **To allow a full statement to act as a subject or object in another statement** (e.g. subjective knowledge, statements that are attributed to a person/source and may be filtered based on that source); example: *Susana thinks that Robert works at UBB*.
* **Nesting is also possible**, example: *John thinks that Patrik lives in Vienna since 2005*.

Diagram

Description automatically generated

Before RDF-star, anonymous nodes were used as a workaround for such patterns, leading to various complications in queries e.g.:

* the decomposition of a statement, followed by its grouping under an anonymous node ("reification"):

:Susana :thinks [rdf:subject :Robert; rdf:predicate :worksAt; rdf:object :UBB].

[rdf:subject :Robert; rdf:predicate :worksAt; rdf:object :UBB] :since 2000 .

* a grouping of multiple properties around an anonymous node interpreted as a complex relationship ("n-ary relationship"):

:Patrik :livesIn [:inPlace :Vienna; :since 2005].

[:connectedCities :Cluj, :Bucharest; :distance [:by :plane; :value 330], [:by :car; :value 450]].

RDF-Star introduces a new delimiter that **converts a statement into a node**, thus allowing for a more natural syntax (whose implementation also has superior performance). With RDF-:

:Susana :thinks <<:Robert :worksAt :UBB>>.

<<:Robert :worksAt :UBB>> :since 2000 .

<<:Patrik :livesIn :Vienna>> :since 2005 .

<< <<:Cluj :connectedTo :Bucharest>> :distance 330>> :by :plane.

<< <<:Cluj :connectedTo :Bucharest>> :distance 450>> :by :car.

Combining the new and the old way it's still possible (and sometimes convenient to avoid nesting, which may complicate some queries):

<<:Cluj :connectedTo :Bucharest>> :distance [:by :plane; :value 330], [:by :car; :value 450] .

**Querying RDF-Star**

Add the following RDF-Star statements to your database (in the Import window make sure that you specify as upload format **Turtle\*** and not simply Turtle, otherwise the non-standard syntax will raise errors!):

<< :Arnold :wasGovernorOf :California >> :from 2003; :until 2011 .

<< :TimBurton :workedWith :JackNicholson >> :commonProjects :Batman.

<< << :JamesCameron :marriedTo :LindaH >> :from 1997 >> :accordingTo <http://wikipedia.org>, :Robert .

<< << :JamesCameron :marriedTo :LindaH >> :until 1999 >> :accordingTo <http://wikipedia.org> .

<< << :JamesCameron :marriedTo :LindaH >> :until 2000 >> :accordingTo :Robert .

:Robert :believes << :Arnold :wasGovernorOf :Austria >>.

:Andrei :believes << :Arnold :wasGovernorOf :California >>.

<< :Arnold :wasGovernorOf :Austria >> :value false .

:FSEGAMoodleSystem :hasPermissions <<:Robert :canEdit :JavaCourse>>,

<<:Robert :canEdit :HTMLCourse>>,

<<:Robert :canRead :DatabaseCourse>>,

<<:Andrei :canRead :JavaCourse>>.

* the first example establishes a timeframe for the governor quality of Arnold (data attached to a relationship);
* the second expresses the collaboration between Tim Burton and Jack Nicholson, then attaches further details on the collaboration itself (the projects where they collaborated);
* the last three add some nesting ("statements about statements about statements"): first a timeframe is established (for the marriage between James Cameron and Linda), then the timeframe is attributed to some sources (wikipedia and Robert); these timeframes are slightly contradicting – both sources agree that the marriage started on 1997, but they disagree on the ending year.

Note: Inside the << >> delimiters it is possible to embed only a single statement (**which will be considered a single node**), therefore in the nested example it's not possible to avoid the repetition by using Turtle-style grouping of properties:

<< <<:JamesCameron :marriedTo :LindaH>> **:from 1997; :until 1999**>> :accordingTo <http://wikipedia.org> .

It is, however possible to apply Turtle grouping outside the RDF-star delimiters:

<< <<:JamesCameron :marriedTo :LindaH>> :from 1997>> :accordingTo **<http://wikipedia.org>, :Robert** .

This convenience is assimilated by the SPARQL language. It can be best understood when starting from exploratory queries:

ASK {**:TimBurton :workedWith :JackNicholson**}

* does not find the embedded statement, because it was stored as a node

ASK {**<<:TimBurton :workedWith :JackNicholson>>**}

* raises an error, because ASK needs to look for statements and we gave it a node

ASK {<<:TimBurton :workedWith :JackNicholson>> **?predicate ?object**}

* finally works because it looks for the whole (meta)statement containing the embedded statement

ASK {<<?x ?y ?z>> :from 2003}

* also works, checks if there are embedded statements valid from 2003

Similarly we can see this in queries that expect to receive a node:

DESCRIBE :LindaH

* does not display the marriage (DESCRIBE does not return connections from embedded statements)

DESCRIBE <<:JamesCameron :marriedTo :LindaH>>

* no results, same reason

DESCRIBE {:TimBurton :workedWith :JackNicholson}

* raises an error, because DESCRIBE needs to look for a node and we gave it a statement

DESCRIBE <<:TimBurton :workedWith :JackNicholson>>

* finally works because we gave it **a statement that's actually a node** (we obtain all metadata about this statement) and that node is not hidden in another embedded statement

DESCRIBE :Robert

* works because the node is not hidden in embedded statements; **the returned results include the embedded statements** connected to (attributed) to Robert, even the nested ones!

DESCRIBE ?x WHERE {<<?x :marriedTo :LindaH>>}

* raises an error because WHERE expects a statement pattern but we gave it a node

DESCRIBE ?x WHERE {<< <<?x :marriedTo :LindaH>> :until ?y>>}

* same problem, a node containing nested embedded statements

DESCRIBE ?x WHERE {<< <<?x :marriedTo :LindaH>> :until ?y>> :accordingTo :Robert}

* finally, sees and describes :JamesCameron (but still it does not return his embedded connections)

Now we can transfer this behaviour to SELECT

SELECT ?x ?y WHERE {?x :workedWith ?y }

* no results, the query does not look by default for embedded statements; **we can extract variables from inside embedded statement, but we have to provide complete statement patterns (keeping in mind that <<...>> is seen as a node)**:

SELECT ?x ?y WHERE {<<?x :workedWith ?y>>}

* raises error, since WHERE expects a pattern of statements and we gave it a node

SELECT ?x ?y WHERE {<<?x :workedWith ?y>> ?predicate ?object}

* finally works since a complete statement is now provided in WHERE

SELECT ?x WHERE {?x :commonProjects :Batman}

* SELECT can also return embedded statements, treated as nodes in normal queries!

Similar thinking must be applied to nesting:

SELECT ?x WHERE {<< <<:JamesCameron :marriedTo :LindaH>> :until ?x>>}

* raises an error because WHERE expects statement patterns but we gave it a node; if we need all marriage ending years available, we have to expand until we express a full statement:

SELECT ?x ?source WHERE {<< <<:JamesCameron :marriedTo :LindaH>> :until ?x>> **:accordingTo ?source**}

* if we want all events and their ending years, the following only gives us the end of Arnold's governance, it will not look inside the marriage timeframes because they are embedded:

SELECT ?x ?y WHERE {?x :until ?y}

* to also get the marriage end years, a logical OR must check for both the presence of :until inside an embedded statement and outside one:

SELECT ?x ?y WHERE

{

{?x :until ?y}

UNION

{<<?x :until ?y>> ?property ?object}

}

* **property paths cannot cross from outside an embedded statement inside one** – we cannot see who was married to the director of Terminator like this (the first property is not embedded, the second is):

SELECT ?x WHERE

{

"Terminator" ^:hasTitle/^:directorOf/:marriedTo ?x

}

* however the Turtle-like syntax is able to achieve this by unpacking the nested statements and matching variables regardless if they are in embedded statements or not!

SELECT DISTINCT ?name WHERE

{

"Terminator" ^:hasTitle/^:directorOf ?director.

<< <<?director :marriedTo ?spouse>> ?p ?o>> :accordingTo <http://wikipedia.org>.

?spouse :hasName ?name

}

We'll come back to RDF-star since it has consequences for numerous other features and query types.

1. You can search on Google "DBPedia John McTiernan" to find its subgraph page [↑](#footnote-ref-1)
2. If you want to also see the anonymous nodes, you have to customize the visualization configuration in Visual Graph – Advanced Graph Configurations. The full configurations are available at https://stackoverflow.com/questions/58454440/visual-graph-showing-blank-nodes (actually it's a query that temporarily converts all blank nodes to URIs for the sake of visualization only) [↑](#footnote-ref-2)
3. For a more general way of filtering by datatype, check the DATATYPE function (<https://www.w3.org/TR/sparql11-query/#func-datatype>) [↑](#footnote-ref-3)
4. Technically, the query looks for all things involved in the relationship playedIn with Arnold. Since we don't have entity types declared, at this point the server does not know that they are movies. [↑](#footnote-ref-4)
5. Even SOLR and Elasticsearch, but not in the free version of GraphDB [↑](#footnote-ref-5)
6. The complete list of functions can be consulted at <https://www.w3.org/TR/sparql11-query/#func-strings> [↑](#footnote-ref-6)