Výstup výzkumného projektu

Jiří Resler

Popis projektu

Zadání

Student se zaměří na obdobu klasického facetového filtrování, ve kterém systém zobrazuje facety pro jednotlivé vlastnosti prvků v daném seznamu a umožňuje uživateli filtrovat pomocí hodnot vlastností.

V současné implementaci uživatel interaguje s aplikací takovým způsobem, že klikáním vytváří graf, a cílem je mít možnost filtrovat uzly, které jsou zobrazeny. Je potřeba zanalyzovat možnosti, které uživatelovi nabídneme, a vybrat způsob, jak toho dosáhneme. Je možné filtrování na základě aktuálního stavu grafu řešit tím, že načítáme facety jeho uzlů, a na základě toho budeme moci uživateli nabídnout možnosti filtrování aktuálního grafu. Další způsob je využít dopředu známe konfigurace, kterou si uživatel sám vybere. To sice znamená, že nějaké informace o uzlech v grafu už víme, konfigurace se ale můžou navzájem i kombinovat.

Student vyvine komponentu, která bude vytvářet facety podle grafových vlastností uzlů, kde vlastností není pouze název a stupeň, ale i další vlastnosti uzlů a také cesty, které z uzlů vedou (např. pro uzly typu politik, z nichž vedou cesty do uzlu politická strana nabídne komponenta facet umožňující filtrování uzlů typu politik dle jejich napojení na politickou stranu).

Požadavky na uživatelské rozhraní

- Nástroj musí být schopen uživateli nabídnout sadu facetů, které budou filtrovat množinu zobrazených uzlů

Platforma, technologie

- Vue.js
- TypeScript
- Cytoscape.js
- SPARQL, RDF

Výstup projektu

Zdrojové kódy rozšíření nástroje Knowledge Graph Visual Browser

Frontend

- KGVB frontend - můj fork
- KGVB frontend - zmergovaný repozitář
    - můj pull request

Backend

- KGVB backend - můj fork
- KGVB backend - zmergovaný repozitář
    - můj pull request

Členové týmu LinkedPipes kteří se podílejí nebo podíleli na vývoji Knowledge Graph Visual Browseru:

- **Štepán Stenchlák**
    - **Původní projekt v rámci bakalářské práce**
- **Oskar Razyapov**
    - **Clustering uzlů**
- **Jiří Resler**
    - **Facetové filtrování**
    - **Do projektu jsem přidal tyhle soubory:**
        - Frontend
        - Backend **– tam jsem přidal jednu funkci**

Dokumentace

- **Uživatelská dokumentace**
    - Webstránka
- **Programátorská dokumentace**
    - Repozitář s dokumentací

Vědecký článek popisující rozšíření

1. Faceted filtering theory

1.1. Intro

The knowledge graph visual browser allows faceted filtering. The idea is to present facets to the user and allow them to choose facet values which will then be used to filter the nodes of the graph. Fig. 1 shows how this component looks to the user.

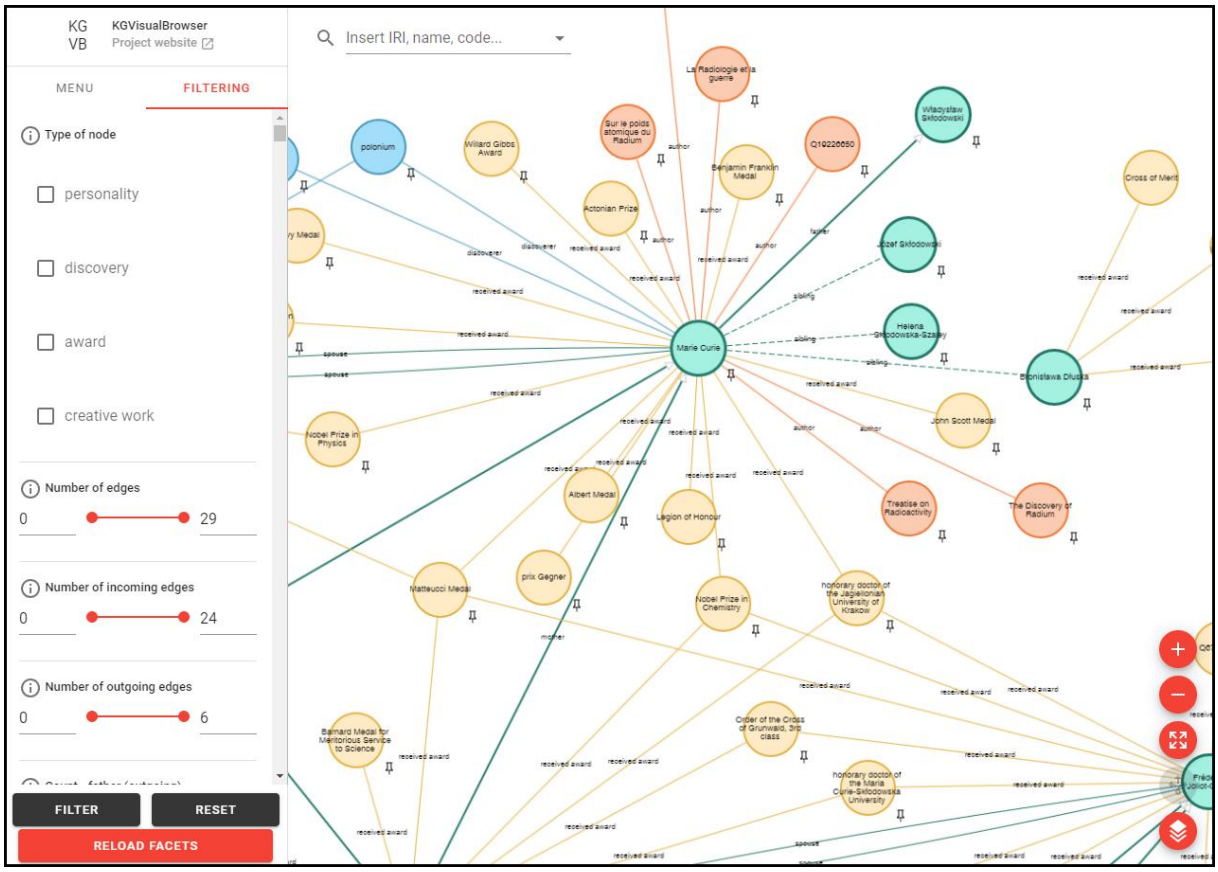

Figure 1: Filtering component

1.2. Usage

At first the facets are loaded and presented to the user, then the user selects facet values they wish to filter by, and then they click the filter button to apply the filtering criteria on the graph – nodes which do not pass the filter are hidden. When they wish to display filtered nodes again and reset facet values to their original state, they can press the reset button. The "reload facets" button can be used when the user wants to make sure that the facets are synchronized with the graph, because under certain circumstances it can happen that facets will contain outdated information about the graph. Later it will be mentioned why that can happen.

1.3. Types of facets

We distinguish two types of facets: facets loaded from a configuration and dynamically (locally) generated facets. This distinction is hidden from the user apart from the fact that facets loaded from a configuration take some time (in seconds) to load.

The second distinction is that a facet can be a "label" facet or a "numeric" facet. A label facet works with labels of resources, for example a facet which allows to filter by first names of people works with people's first names as labels. A label facet is rendered using checkboxes and the user chooses from the given labels. It is called a label facet because most of RDF resources have a human readable label defined, but it can work with any string. For example, it can display dates of births of people in the graph and the user can filter by those values. A numeric facet, on the other hand, works with numbers. It is rendered using a range slider and

the user chooses a range of a facet to filter by. For example, a numeric value for a node can be its degree. The user can then select a closed interval of values which can then be used to filter the graph.

1.4. Loading and generating facets

Facets are generally being loaded or generated when the filtering component is mounted in the application, or when there was an expansion, or when the "reload facets" button is clicked. The tool is able to search for facets but it does not discard facets which have low value for the user. The result of that is that there may be too many facets generated which may not be pleasant for the user. There is room for improvement, and it can be addressed in the future.

1.5. Updating facets

Facets are updated as the graph evolves. After an expansion of nodes is completed, facets are loaded and computed for newly added nodes, and recomputed for nodes for which facet values could also be affected by the expansion. For example, a facet about number of edges of nodes needs to be updated for the node which was the source of the expansion, and we also need to compute it for the added nodes.

Facets are also updated when a node or more nodes are deleted. Not only is the node itself deleted so we need to update all facets' values, but it may be that the deleted node had edges attached to it. Deletion of a node can then affect some of the locally generated facets, so they need to be recomputed. It needs to be done for nodes which were not deleted but one of their edges got removed. For example, a facet about number of edges of nodes needs to be updated for the node which got its edge removed.

1.6. Facets loaded from a configuration

The creator of a configuration can define facets which they want to enable users to filter by. For example, they can create a facet which will enable users to filter by a person's first (given) name or by their country of birth. A facet can be created within a configuration where it is used, or it can be referred to by its IRI – it can be reused in another configuration. These facets operate on a given dataset, so the creator of a facet needs to have knowledge of relationships between nodes in the dataset. For example, a node in a dataset can have an edge called "date of birth". If a facet would be defined correctly, the tool would be able to extract this piece of information about all nodes in a visual graph, allowing for filtering by this facet. But we do not have to stop there. The creator of a facet uses SPARQL to obtain information from a dataset, so possibilities of facets are not limited in any way. Another facet can for example be that a node of type person can have an edge referring to their place of birth, which can be another node of type country, and it can have an edge of type "population size". So, the facet would be used to filter by the population sizes of countries where people were born in.

Fig. 2 shows a conceptual model of a configuration which has facets defined.

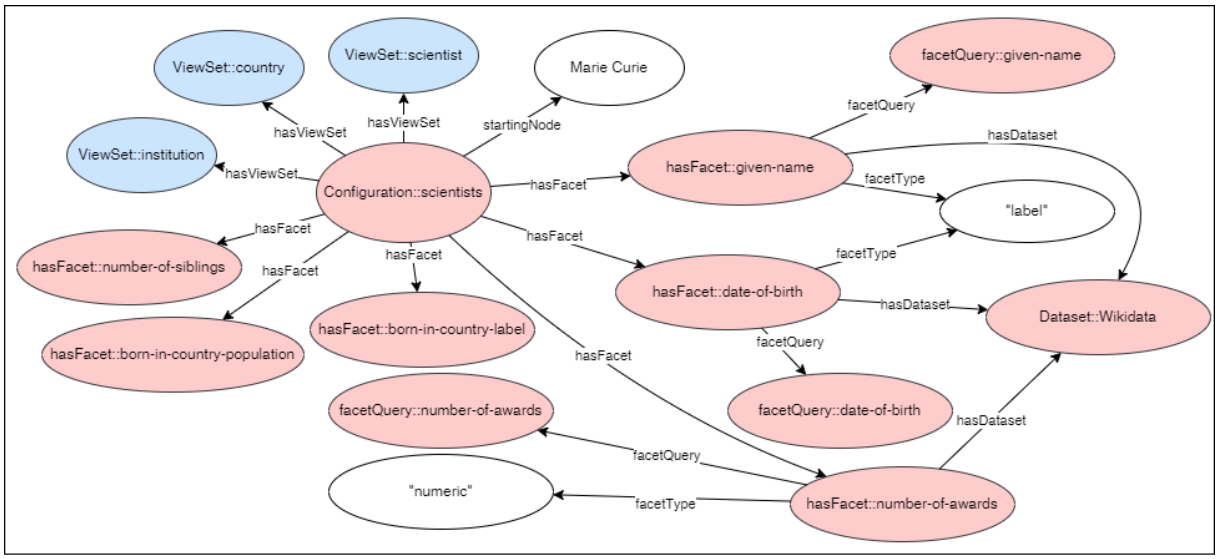

Figure 2: Visual configuration for scientists with defined facets

1.6.1. How to define a facet

Facets loaded from a configuration are defined like shown in fig. 3. The creator of a configuration can set a facet for a configuration using the *browser:hasFacet* predicate followed by a facet's IRI. A definition of a facet contains its title as *dct:title* and description as *dct:description* which are used to inform the user about what the facet means. The *browser:facetType* can have two values - either a facet is a "label" facet or a "numeric" facet. The browser:hasDataset specifies a dataset on which the facet's query will be executed. The dataset needs to have a SPARQL endpoint available so that it is possible to run queries on it. The browser:facetQuery is a SPARQL query which returns nodes' IRIs and values associated with them given that they meet the query's criteria. The only two requirements for a facet SPARQL query are that it must return RDF triples in this format: node's IRI :: arbitrary predicate name :: a label or a numeric value, and that it must return either only labels or only numeric values in the third part of the triple format. Names of variables in the CONSTRUCT block do not matter. The placeholder #INSERTNODES in a query serves for the backend function so that it can insert nodes' IRIs which it gets sent from the frontend. Here is a runnable example of a facet query from fig. 2. The query in fig. 2 contains this path: wdt:P735/rdfs:label. The wdt:P735 property is the wikidata's "given name" property of a person and the rdfs:label is the name's string representation.

```
<https://linked.opendata.cz/resource/knowledge-graph-browser/facet/given-name> a browser:Facet ;
  dct:title "First name"@en;
  browser:facetType "label";
  dct:description "Facet to filter people based on their given names."@en;
  browser:hasDataset <https://linked.opendata.cz/resource/knowledge-graph-browser/dataset/wikidata>;
  browser:facetQuery """
    PREFIX wdt: <http://www.wikidata.org/prop/direct/>
    PREFIX browser: <https://linked.opendata.cz/ontology/knowledge-graph-browser/>

    CONSTRUCT {
      ?node browser:queryPath ?targetNode.
    } WHERE {
      #INSERTNODES
      ?node wdt:P735/rdfs:label ?targetNode.

      FILTER (LANG(?targetNode) = "en")
    }
    """.
```

Figure 3: Definition of a label facet

1.6.2. Examples of facet definitions

Figures 4 and 5 show more definitions of facets. The facet in fig. 4 allows to filter by number of siblings of people. The wdt:P3373 property is the wikidata's sibling property of a person.

```
<https://linked.opendata.cz/resource/knowledge-graph-browser/facet/number-of-siblings> a browser:Facet ;
  dct:title "Number of siblings"@en;
  browser:facetType "numeric";
  dct:description "Facet to filter people based on how many siblings they have."@en;
  browser:hasDataset <https://linked.opendata.cz/resource/knowledge-graph-browser/dataset/wikidata>;
  browser:facetQuery """
    PREFIX wd: <http://www.wikidata.org/entity/>
    PREFIX wdt: <http://www.wikidata.org/prop/direct/>
    PREFIX browser: <https://linked.opendata.cz/ontology/knowledge-graph-browser/>

    CONSTRUCT {
        ?node browser:queryPath ?targetNode.
    } WHERE {
    {
        SELECT ?node (COUNT(?sibling) AS ?targetNode)
        WHERE {
            #INSERTNODES
            ?node wdt:P3373 ?sibling.
        }
        GROUP BY ?node
    }
    }
    """.
```

Figure 4: Number of siblings facet definition

The facet in fig. 5 allows filtering by population sizes of countries which people were born in. The wdt:P19 property is wikidata's "place of birth" property (it points to a city). The wdt:P17 property is wikidata's "country" property. The wdt:P1082 property is wikidata's "population" property.

```
<https://linked.opendata.cz/resource/knowledge-graph-browser/facet/born-in-country-population> a browser:Facet ;
  dct:title "Born in country with population"@en;
  browser:facetType "numeric";
  dct:description "Facet to filter people based on the population of the country they were born in."@en;
  browser:hasDataset <https://linked.opendata.cz/resource/knowledge-graph-browser/dataset/wikidata>;
  browser:facetQuery """
  PREFIX wdt: <http://www.wikidata.org/prop/direct/>
  PREFIX browser: <https://linked.opendata.cz/ontology/knowledge-graph-browser/>

  CONSTRUCT {
    ?node browser:queryPath ?targetNode.
  } WHERE {
    #INSERTNODES
    ?node wdt:P19/wdt:P17/wdt:P1082 ?targetNode.
  }
    """.
```

Figure 5: Born in country with population facet definition

1.7. Dynamically generated facets

Dynamically generated facets are facets which are found locally based on the current state of the graph. They are not specified in any configuration. The principles of these facets are the same for every graph no matter what configuration is chosen. These are for example graph properties of nodes like their degree etc.

1.7.1. Type of node facet

Every node has a type, and this facet allows users to filter by all types of nodes which are present in the graph. Fig. 6 shows an example of values of the type facet.

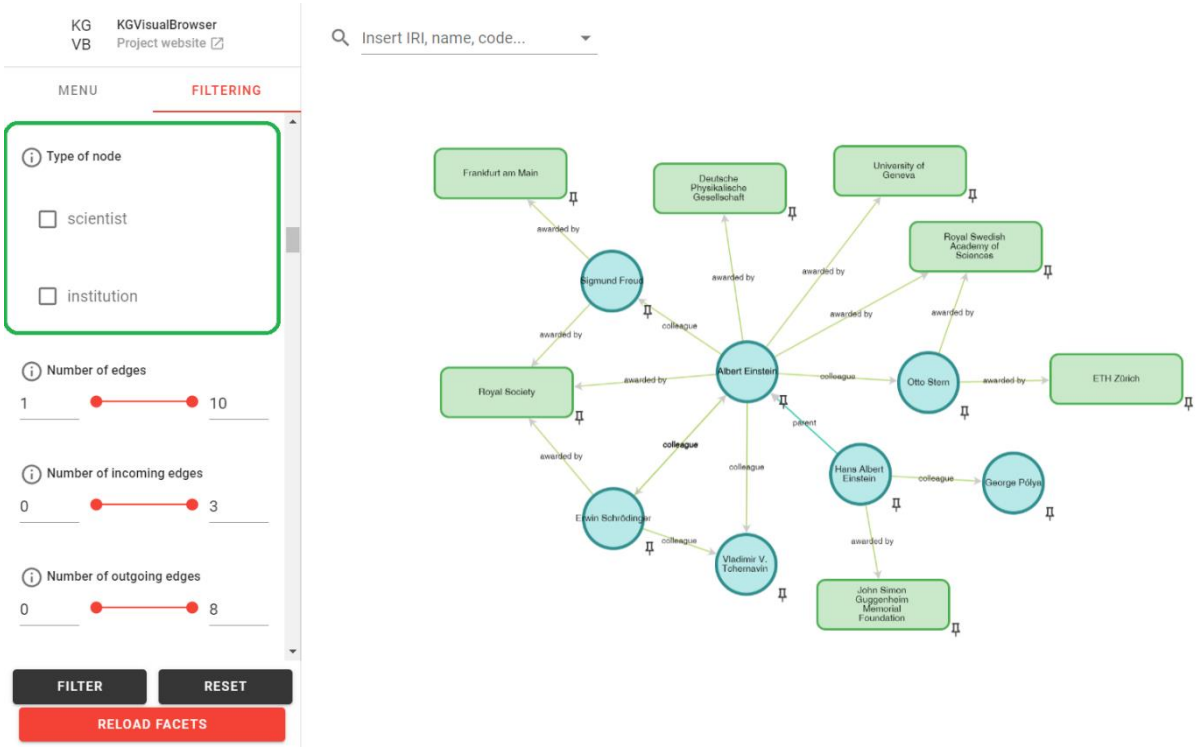

Figure 6: Example values of the type facet

1.7.2. Number of edges facets

These facets allow filtering by the total number of edges of nodes as well as number of incoming and outgoing edges of nodes. Fig. 7 shows examples of number of edges, number of incoming edges and number of outgoing edges facets.

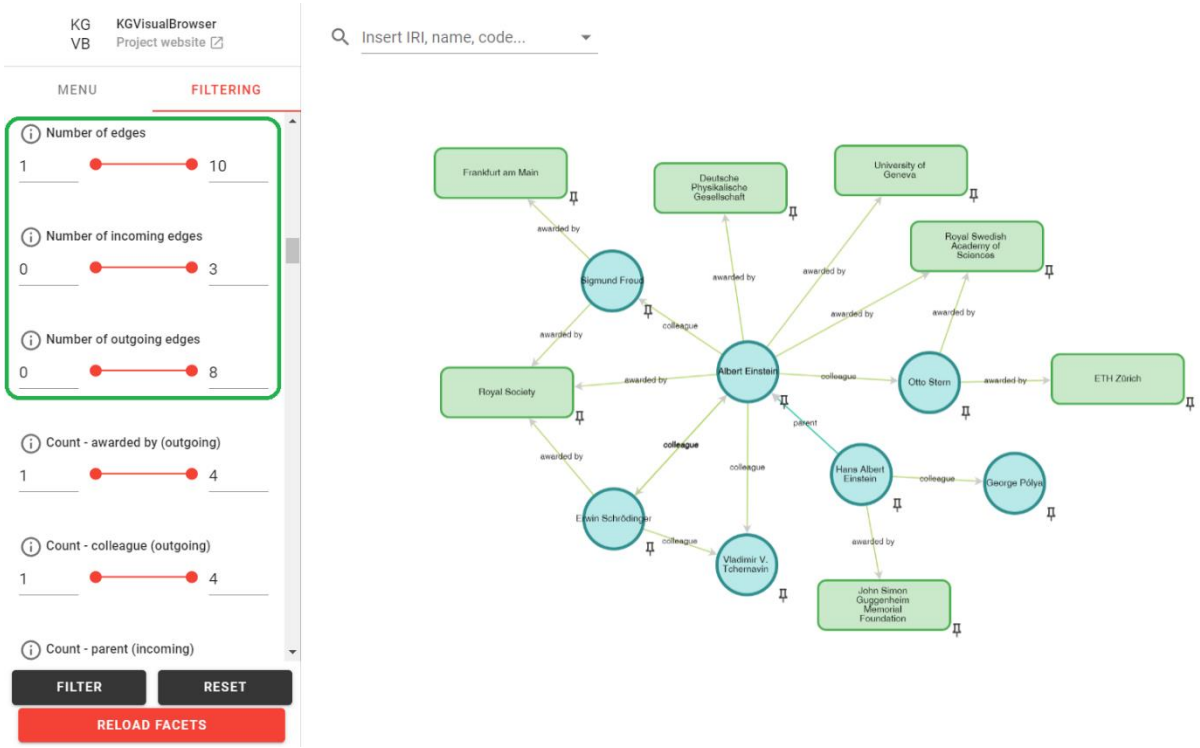

Figure 7: Example of number of edges facets

1.7.3. Number of edges of a certain type facets

This facet filters nodes by their number of edges of a specific type. This means that a node can have different types of edges and a certain facet is based on a certain type of edge which appears in the graph. For every type of edge there is a corresponding facet found. We also differentiate the direction of an edge, because it is a big part of the meaning of the relationship and not all relationships in the graph must be reflexive. Fig. 8 shows examples of these facets.

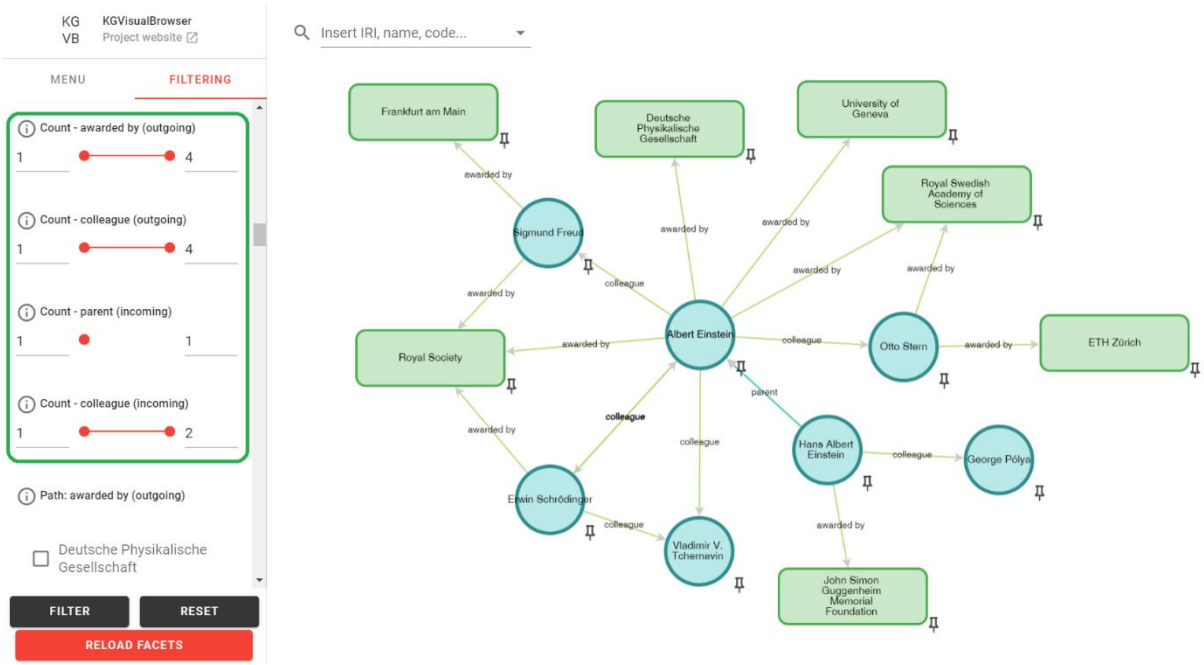

Figure 8: Examples of count of edges of certain types facets

1.7.4. Path facets

This type of facet is specified by a path in the graph which means that it expresses a more complex relationship. For example, a facet can be specified by this path: "colleague (outgoing edge)" / "awarded by (outgoing edge)" and it means that a node (in this case of type scientist) has a colleague who was awarded by some institution. The user can filter nodes based on whether there exists such a path which starts from those nodes. On top of that can the user also select what label must the node at the end of that path have, and thus it is a label facet. To conclude the example, the user is able to filter nodes based on whether scientists have colleagues who are awarded by institutions which the user selects. Fig. 9 shows an example of a path facet.

There can be many paths like this in a graph. Paths we are interested in are either of length 1 or 2. We think that paths of lengths 3 or more would not be very useful to the user. The depth-first search algorithm is used to search for paths in the graph. Let's say a new node was added after an expansion. We want to update facets so that they are synchronized with the graph. The DFS algorithm is run from the node and finds all paths of lengths 1 and 2. When a facet of a particular path already exists, it is updated.

When a node is deleted and was a source for finding a path facet, it can happen that a facet will contain outdated information about the graph. When a path facet is being found, the depth-first search can traverse through multiple edges and currently the tool does not support the meanings to be able to know where the facet's path originated. So, if a node is deleted it does not mean that the facet is updated correctly. In other words, deletion is not inverse to expansion in terms of updating facets. That is why we need the "reload facets" button.

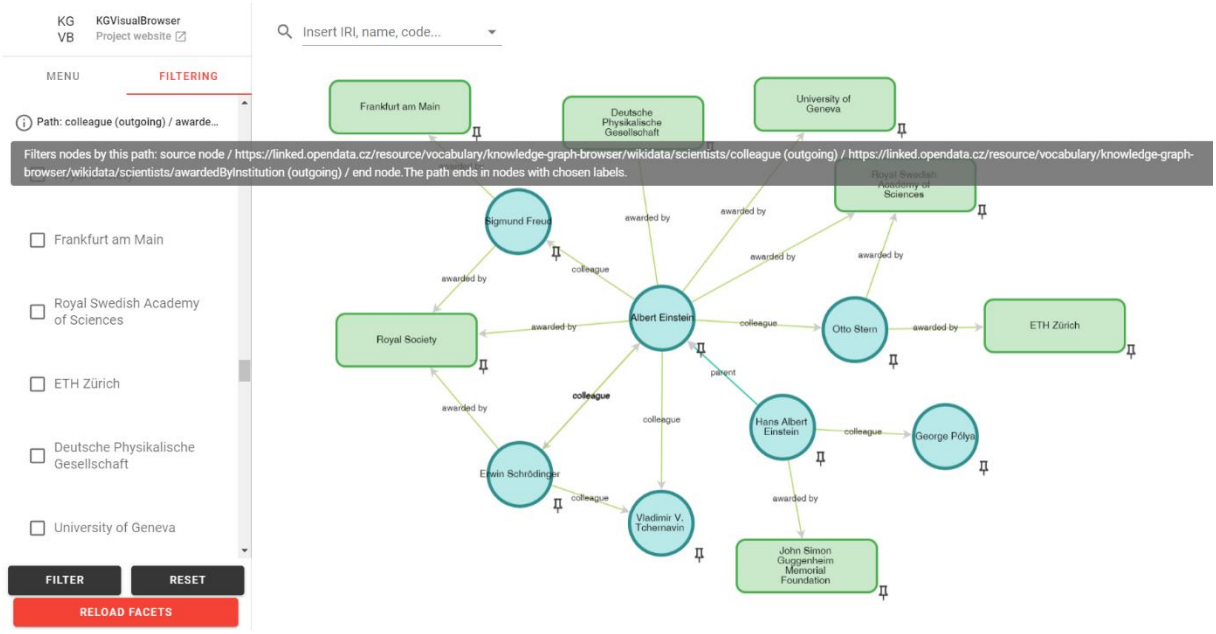

Figure 9: Example of a path facet

2. Implementation

2.1. Overall architecture

The following picture depicts the Knowledge Graph Visual Browser's architecture after faceted filtering was added:

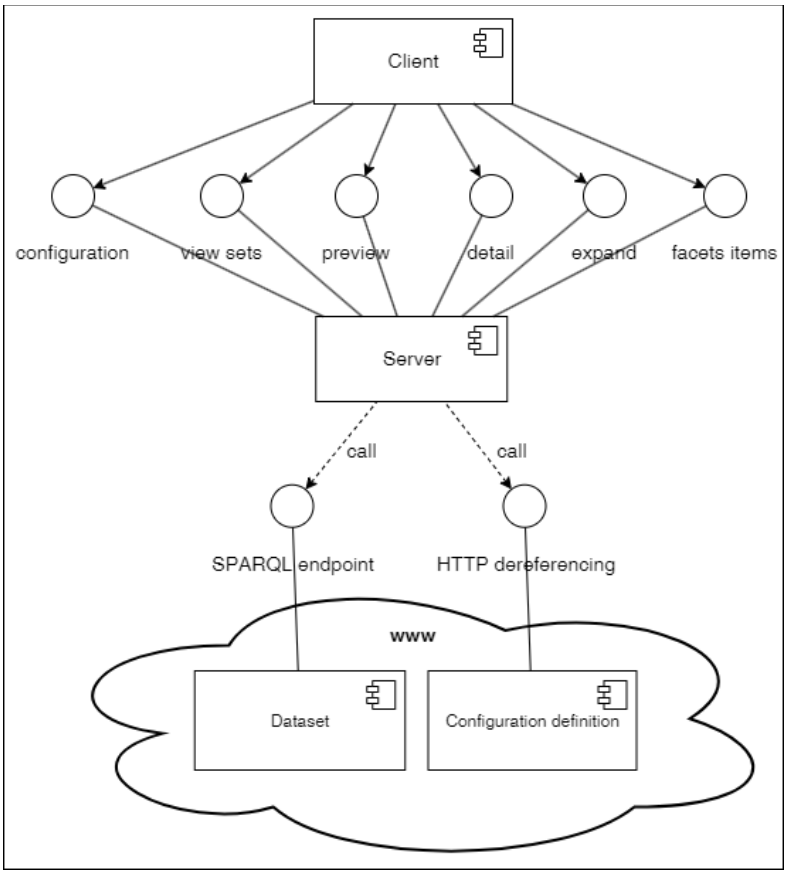

Figure 10: The Knowledge Graph Visual Browser's architecture

2.2. Frontend

The client contains code for rendering facets on the screen and it works with data that get sent to it by the KGVB server. Facets loaded from a configuration are treated the same way as locally generated facets and they are indistinguishable for the user. If no facets are defined in a configuration, then only locally generated facets will be found.

The following sequence diagram shows how the components interact with each other during runtime when the tool is loading configuration facets:

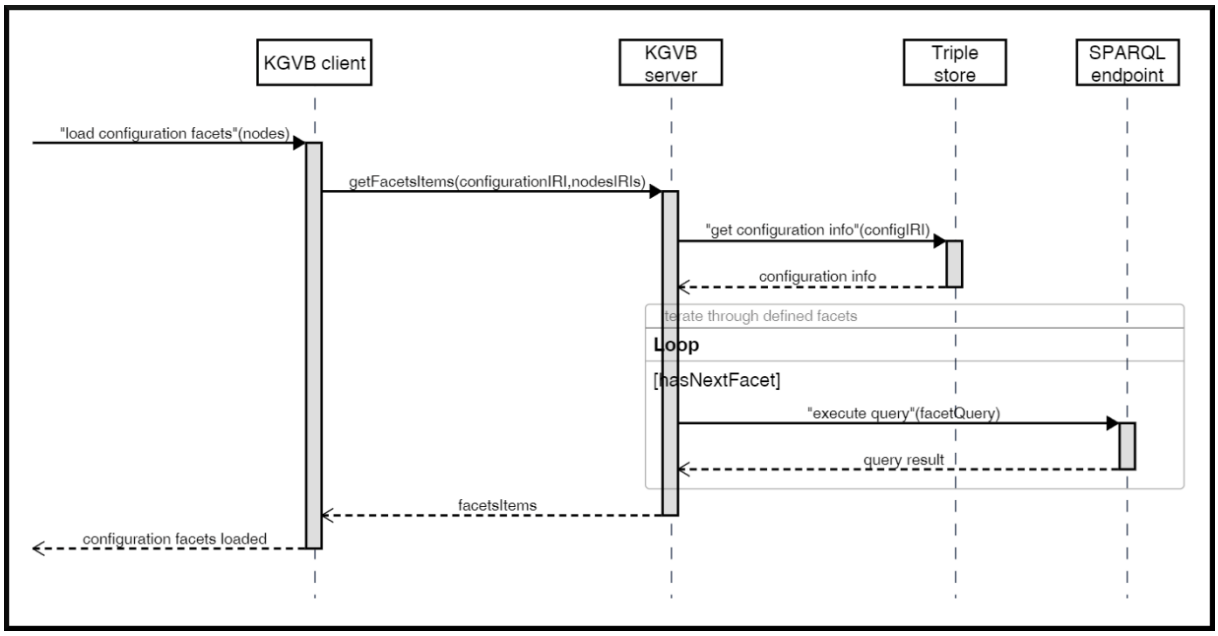

Figure 11: The process of loading configuration facets

At first the client wants to load facets, both from the graph's configuration and dynamically generated. It is performed when the user clicks the filtering tab for the first time (when the component is mounted), or after an expansion for newly added nodes, or when the user clicks the "reload facets" button. The KGVB frontend calls the getFacetsItems function on the KGVB server, whose inputs are the graph's configuration's IRI and nodes' IRIs for which the facets are supposed to be loaded. The server then loads the configuration's definition as a set of triples. For each defined facet in the configuration the server loads information about the facet and sends its defined query to a SPARQL endpoint. The endpoint is specified in the configuration. When the KGVB server receives a response from a SPARQL endpoint, it parses the response and sends the result to the frontend. The client then creates a list of facets which it shows to the user.

2.2.1. Indexing and filtering

Filtering is performed on the client side of the application. For this purpose does every facet have its own index for storing nodes' values. Indexes for facets are created when facets are created and updated as the user adds or removes nodes to the graph. When an index becomes empty because all nodes it was indexing were deleted, its facet is marked as undefined and won't be rendered. There is a difference between label and numeric facets in that how nodes are indexed for filtering. A label facet has a map as its index where keys are

labels (for example given names of people) and values are IRIs of nodes (for example people) which have the given property (for example the key “Albert” would have an array containing the IRI of Albert Einstein associated with it). Here is an example of a label facet's index:

```
{
   "Albert" => ["IRI_1"],
   "Marie" => ["IRI_2", "IRI_3"]
}
```

A numeric facet has an array for its index and its elements are nodes with their numeric values associated to them. For example, the index of a facet about number of edges of nodes would contain IRIs of nodes and for each node its number of edges. Here is an example of a numeric facet's index:

```
[
  {
    "nodeIRI":"IRI_1",
    "value":5
  },
  {
    "nodeIRI":"IRI_2",
    "value":7
  }
]
```

When the user clicks the filter button, a set of nodes that pass all selected criteria is gathered and all nodes of the graph are tested if they are present in this set. If a node doesn't pass the filter, it will become hidden.

2.2.1.1. Example of indexing and filtering

Let's consider the following graph for an example:

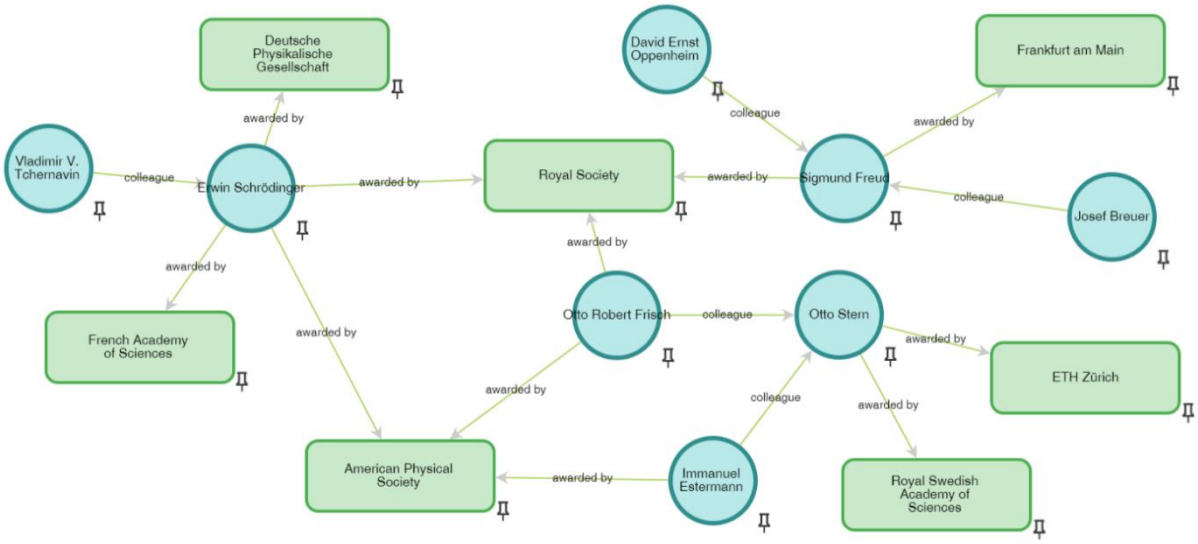

Figure 12: Example of a visual knowledge graph

The facets we will examine will be:

- Born in country facet - every person has a country they were born in
- Number of outgoing edges
- Path facet defined by this path: colleague (outgoing) / awardedByInstitution (outgoing)
- This facet means that there must lead a path like this from a node which ends in nodes with labels chosen by the user

Let the configuration have only one facet defined - the born in country facet which allows to filter by countries which people were born in. Its query finds countries where people in the graph were born and returns triples like this: node (person) :: query path (only people who have their place of birth defined are returned) :: country.

So, the client sends a request to the server which will return facet items. The client will create a facet (if it wasn't created before) and create (or update) its index with the following content (the application works with nodes' IRIs, but here we use human readable names of nodes for clarity):

```
{
   "Austria" => ["Erwin Schrödinger", "Otto Robert Frisch", "Josef Breuer"],

   "Germany" => ["Immanuel Estermann"],

   "Czech Republic" => ["David Ernst Oppenheim", "Sigmund Freud"]
 }
```

We can calculate rest of the facets on the client from information in the graph. The number of outgoing edges facet's index will have this content:

```
[
   {
      "nodeIRI":"Erwin Schrödinger",
      "value":4
   },
   {
      "nodeIRI":"Otto Robert Frisch",
      "value":3
   },
   {
      "nodeIRI":"Sigmund Freud",
      "value":2
   },
   {
      "nodeIRI":"David Ernst Oppenheim",
      "value":1
   },
   {
      "nodeIRI":"Josef Breuer",
      "value":1
   },
   {
      "nodeIRI":"Immanuel Estermann",
      "value":2
   },
   {
      "nodeIRI":"Vladimir V. Tchernavin",
      "value":1
   },
   {
      "nodeIRI":"Otto Stern",
      "value":2
   }
]
```


The path facet's index will contain these information:


```
{
   "Deutsche Physikalische Gesellschaft" => ["Vladimir V. Tchernavin"],

   "Royal Society" => ["Vladimir V. Tchernavin", "David Ernst Oppenheim", "Josef
Breuer"],

   "Frankfurt am Main" => ["David Ernst Oppenheim", "Josef Breuer"],

   "French Academy of Sciences" => ["Vladimir V. Tchernavin"],

   "American Physical Society" => ["Vladimir V. Tchernavin"],

   "Royal Swedish Academy of Sciences" => ["Vladimir V. Tchernavin"],

   "ETH Zürich" => ["Otto Robert Frisch", "Immanuel Estermann"]
 }
```


Let's say that the user chooses these values for facets:

- Born in country facet: labels Austria, Germany, Czech Republic

- Number of outgoing edges facet: range <1, 2>
- The path facet defined by this path: colleague (outgoing) / awardedByInstitution (outgoing): labels Royal society, Deutsche Physikalische Gesellschaft, Royal Swedish Academy of Sciences

When the filter button is pressed, each facet returns a set of nodes which pass the selected criteria.

The born in country facet will return this set:

```
{"Erwin Schrödinger", "Otto Robert Frisch", "Sigmund Freud", "David Ernst Oppenheim", "Josef Breuer", "Immanuel Estermann"}
```

The number of outgoing edges facet will return this set:

```
{"Vladimir V. Tchernavin", "Sigmund Freud", "David Ernst Oppenheim", "Josef Breuer", "Immanuel Estermann"}
```

The path facet will return this set:

```
{"Vladimir V. Tchernavin", "Otto Robert Frisch", "David Ernst Oppenheim", "Josef Breuer", "Immanuel Estermann"}
```

We need the nodes to pass all filters, so an intersection of all nodes is made with this result:

```
{"Josef Breuer", "David Ernst Oppenheim", "Immanuel Estermann"}
```

This is a set of nodes which should remain visible, so if a node is not present in this set, it will be hidden.

2.3. Backend

The GET facet items function from fig. 10 works as an intermediate between the KGVB frontend and RDF datasets. Its inputs are the graph's configuration's IRI and nodes' IRIs for which the facets are supposed to be loaded. It reads a query from a facet definition (if there is any facet defined in a configuration), inserts nodes' IRIs from frontend into that query and sends it to a SPARQL endpoint. Then it parses its response and sends the result to the frontend. Here is an example of what a facet's query can look like after the function inserts nodes' IRIs which it gets sent from the frontend. When it is run there is also an example of a SPARQL endpoint response which the backend function parses.

3. Performance tests and evaluation

3.1. Performance tests

The implementation tests can be divided into two parts. The first part is loading and rendering of facets and the second part is filtering. Tests were run on graphs of sizes 50 and 100 nodes.

3.1.1. Loading and rendering facets

Loading facets from a configuration takes some time to complete as the KGVB client and server wait for an RDF dataset's response. Computing facets locally is much faster, and so graphs whose configurations do not have any facets defined have their facets loaded sooner. Another important part is how long it takes to render the facets to the screen.

For a graph of size 50 nodes the tests showed these results:

Action	Time
Server response	4515 ms
Computing on client	18 ms
Rendering	4759 ms

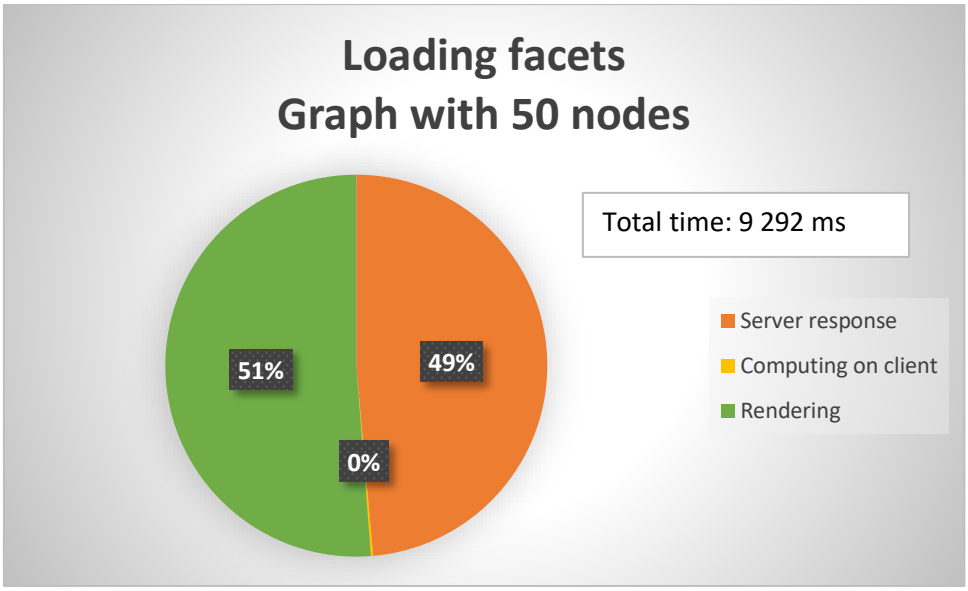

Figure 13: Results of loading facets on a graph with 50 nodes

For a graph of size 100 nodes the tests showed these results:

Action	Time
Server response	4607 ms
Computing on client	69 ms
Rendering	8718 ms

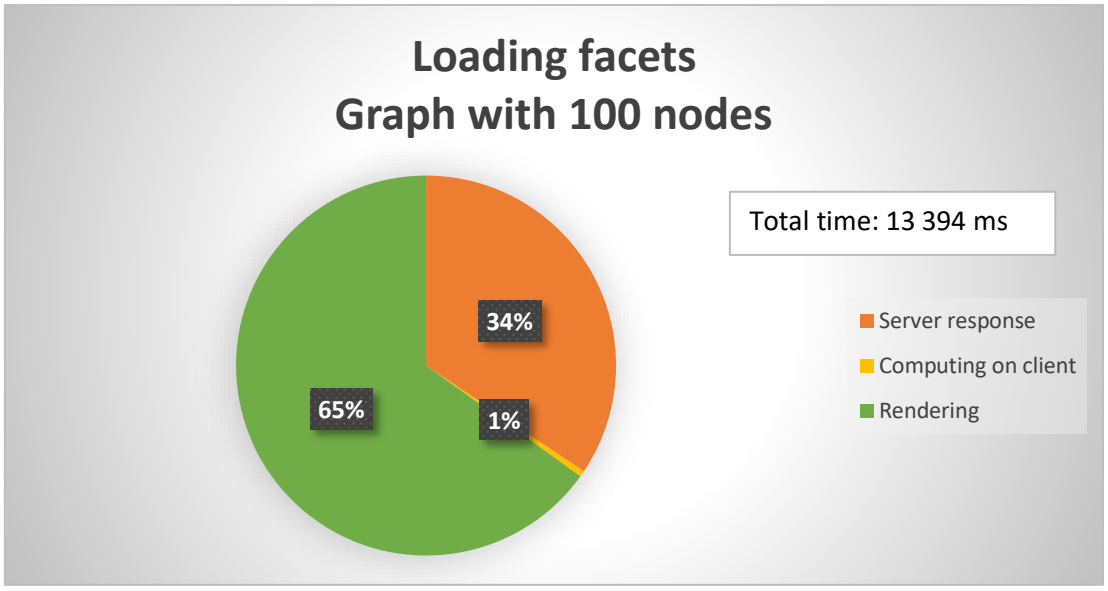

Figure 14: Results of loading facets on a graph with 100 nodes

3.1.2. Filtering

Filtering is implemented on the client so there is no time spent waiting for the server. The only challenge here is how long it will take the graph visualization library to hide filtered nodes and rearrange the remaining ones.

Tests revealed that filtering a graph with 50 nodes takes this long:

Action	Time
Computing on client	0.94 ms
Rendering	2361 ms

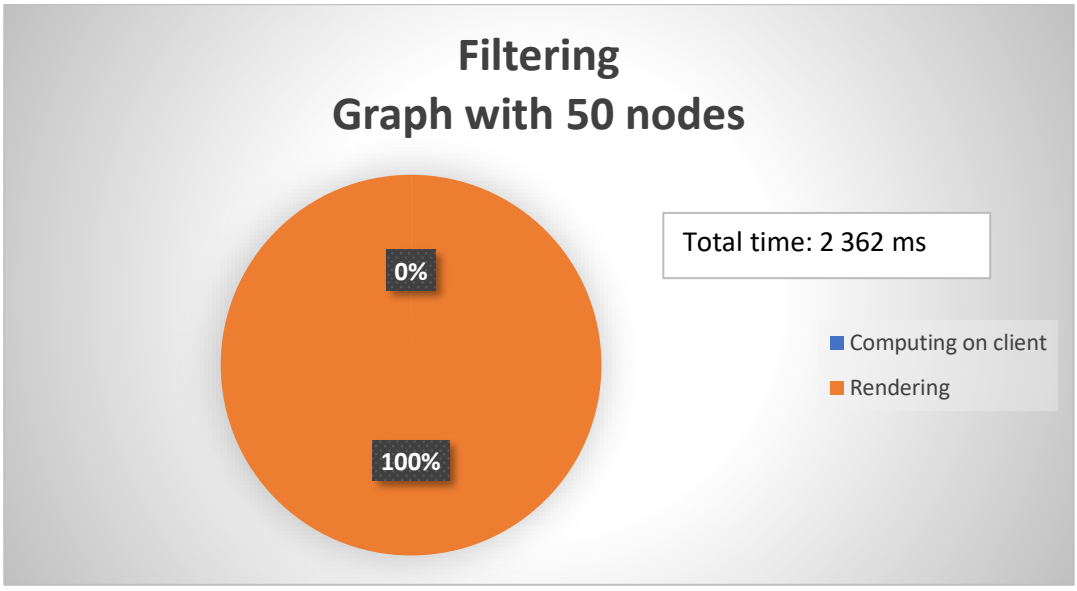

Figure 15: Result of the filtering test on a graph with 50 nodes

Testing filtering on a graph with 100 nodes ended up with these results:

Action	Time
Computing on client	2.3 ms
Rendering	3929 ms

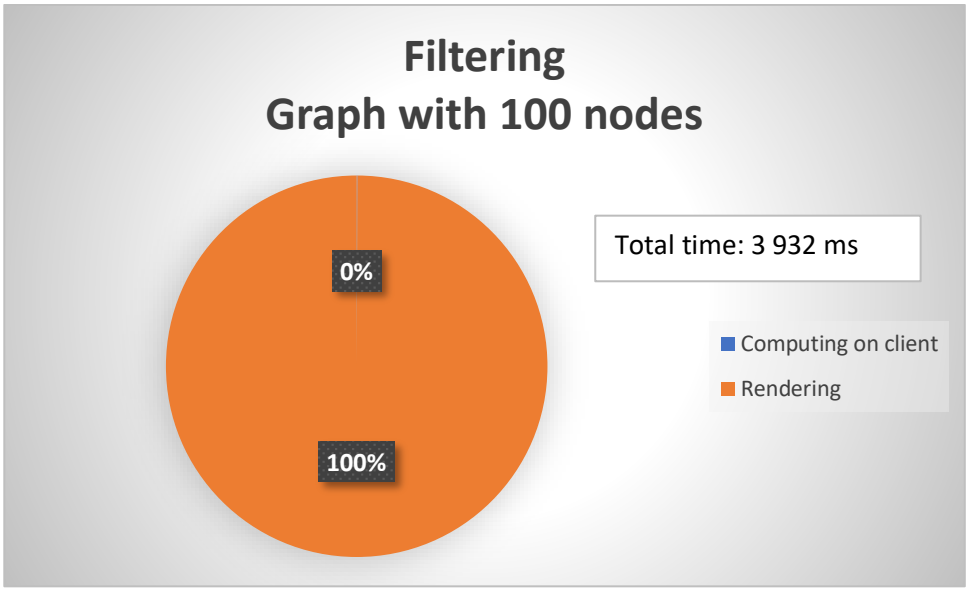

Figure 16: Result of the filtering test on a graph with 100 nodes

3.2. Evaluation of the test results

The tests showed that the implementation is quite slow even for a small graph with 100 nodes. There are two reasons why that is. The first reason is that the client is rendering all facets and their values at once which is a big amount of HTML elements. The solution to this problem could be to use a third-party component which would render only elements which are visible for the user.

The second reason is that the graph visualization library which the tool uses is slow when it is hiding nodes and rearranging nodes which should remain visible after filtering. The reason is that it uses complex calculations to do it. It is most noticeable when the user is using the default layout option. Choosing a simpler layout strategy will speed things up.

The tests were done only for graphs of sizes up to 100 nodes because the main user scenario is that the user starts with only one node and will never create a larger graph (than the one we tested the implementation on) by expanding nodes.