Query Interfaces and Class Hierarchies

Nina Gjersøyen Løkamoen

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Institute for Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2022

Query Interfaces and Class Hierarchies

Nina Gjersøyen Løkamoen

Abstract

OptiqueVQS is a visual query system, which makes it possible for users to create queries without having knowledge of the underlying query language. A problem for OptiqueVQS is that it does not currently have a good way to handle information about subclasses.

In this thesis we will present four different options for how the interface can be changed to better support subclass selection, and make a choice to implement one of them based on analysis and feedback from a user study. We will also make some changes to the backend, which we believe will improve the logic for which options are shown on the frontend.

After making changes to both the backend and the frontend, we will conduct a usability study where we evaluate our new implementation.

Acknowledgements

First and foremost, I would like to thank my two supervisors, Martin Giese and Ahmet Soylu, for all their help and support. They have done a great job guiding me, and have always found a way to help me progress when I felt stuck. I would also like to thank my boyfriend Mayo for all his support and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis we will present a visual query system called OptiqueVQS. This system which makes it possible for users to create queries without having knowledge of the underlying query language. A problem for OptiqueVQS is that it does not currently have a good way to handle information about subclasses.

To figure out how to best present options for subclass selection, we will come up with several different potential solutions for the interface, and create wireframes to represent them. We will then conduct a small user study and do an analysis of the solutions based on feedback and interactions. We will then choose one of the solutions to implement.

We will also make some changes to the backend, which we believe will improve the logic for which options are shown on the frontend.

After making changes to both the backend and the frontend, we will conduct a usability study where we evaluate our new implementation.

1.1 Research question

How to present and handle information about subclasses in a visual query system, such as OptiqueVQS?

1.2 Contributions

1. For the backend, we have come up with, and implemented, an alternative way of deciding which options should be shown on the frontend.

2. For the frontend, we have presented four possible solutions for handling subclass selection, and done an analysis for each of them, including a user study.

3. We implemented one of the proposed solutions, and conducted a usability study for the final version.

1.3 Structure

In chapter 2 we give a brief introduction to some relevant concepts, and describe the basic usage of OptiqueVQS.

In chapter 3 we provide a more detailed problem description.

In chapter 4 we will talk about how the backend workds, and how we can improve it.

In chapter 5 we will present four possible solutions for the interface, and do a user study with wireframes to help decide which of the solutions to implement.

In chapter 6 we will talk about how we implemented the chosen solution from the previous chapter.

In chapter 7 we will evaluate the finished product with a user study and discuss the results.

In chapter 8 we will talk about aspects of the solution that can be improved further and discuss some open questions.

In chapter 9 we will present our conclusion.

Chapter 2

Preliminaries

In this chapter we will give a short introduction to some of the concepts that are relevant for this thesis. We will also give an introduction to OptiqueVQS, and describe basic usage of the system.

2.1 RDF

RDF stands for Resource Description Framework, and is a framework for representing information about resources on the Web [11]. These resources can be anything, for example people, physical objects or abstract concepts [12]. With RDF we can make statements about resources. These statements express a relationship between two resources, and are always expressed in triples. Each triple consists of a subject, a predicate (also called a property) and an object, and has the following structure:

```
<subject> <predicate> <object>
```

The subject and the object are the two resources that are related, and the predicate represents the relationship between the two resources. Let's look at some example triples:

```
<Nina> <is a> <person>
<Nina> <lives in> <Oslo>
<Oslo> <is located in> <Norway>
```

If we look at the middle triple, Nina is the subject, "lives in" is the predicate, and Oslo is the object. In the bottom triple, Oslo is the subject. The fact that a resource can be an object in one triple and a subject in another makes it possible to create and find connections between triples.

A set of triples form an RDF graph, where subjects and objects are nodes, and predicates are edges. The graph is directed, the predicates always go from the subject to the object.

2.2 SPARQL

SPARQL is a query language used to query and manipulate RDF graphs [17]. The results from a SPARQL query are usually shown as a table, and can be presented in four different machine-readable formats: XML, JSON, CSV and TSV. The following definition is a good explanation of how SPARQL queries work [5]:

Definition. *Most forms of SPARQL query contain a set of triple patterns called a basic graph pattern. Triple patterns are like RDF triples except that each of the subject, predicate and object may be a variable. A basic graph pattern matches a subgraph of the RDF data when RDF terms from that subgraph may be substituted for the variables and the result is RDF graph equivalent to the subgraph.*

Let's look at an example of a simple SPARQL query:

```
SELECT ?name
WHERE
{
  ?x type Human .
  ?x hasName ?name .
}
```

In this query we are asking for the names of all humans in our data. We can see that the SPARQL query consists of two clauses; SELECT and WHERE. In the SELECT clause we put the variables we want to see in the results, in this case name. In the WHERE clause we put the basic graph pattern that we want to match against the data graph.

Lets look at a slightly longer SPARQL query:

```
SELECT ?name
WHERE
{
  ?x type Human .
  ?x hasName ?name .
  ?x drives ?y .
  ?y type Vehicle .
  ?y type Car .
}
```

In this SPARQL query we are asking for the names of the humans that drive a vehicle, and the vehicle is a car.

2.3 Ontology

An ontology is defined as "a formal, explicit specification of a shared conceptualization" [16], and can be used to create a formal model that describes the structure of a system [4]. This model includes information about classes and properties, such as class hierarchies for concepts, and domain and range for properties.

2.3.1 OWL 2

OWL 2 (formally OWL 2 Web Ontology Language) is an ontology language that expresses relationships between classes, properties, individuals, and data values for ontologies, and can be used with information written in RDF [18].

Basic statements that are expressed in an OWL ontology are called axioms [8]. For example, we can use a subclass axiom to express that Dog is a subclass of Animal. Or, if we have two classes Cat and Dog, and we know that it is impossible for one individual to be both of those classes, we can express this with a disjointness axiom.

We can also use axioms to express domain and range for properties. Let's say we have an object property hasPet. We could use axioms to say that the domain for this object property would be Human, and the range would be Animal.

2.3.2 OWL API

The OWL API is an API that can be used for creating, parsing, manipulating and serialising OWL Ontologies using Java. The newest version of the OWL API is version 5, and all versions from 3.1 and up are designed for OWL 2. [9]

2.3.3 Example ontology

In this section we will present an example ontology. This ontology will be used in examples throughout this entire thesis, and it will also be used in the user studies we do with OptiqueVQS. We chose to create a simple ontology that should be understandable for everyone, and not require any background knowledge. In the next subsections we will present classes, object properties and data properties.

Classes

Here we will present some of the main classes and subclasses in our ontology. A classes' subclasses will be indented right underneath the relevant class. The three main classes are disjoint with each other, and all subclass "siblings" are disjoint with each other.

- Animal
  - Cat
  - Cow
  - Dog
  - Goat

  - +++
- Human
  - Adult
  - Minor
- Place
  - City
  - Forest
  - Town

Object properties

- hasDriversLicense
  - Domain: Adult
  - Range: DriversLicense
- hasOwner
  - Domain: Animal
  - Range: Human
- hasPet
  - Domain: Human
  - Range: Animal
- livesIn
  - Domain: Animal or Human
  - Range: Place

Data properties

- hasAge
  - Domain: Animal or Human
- hasPopulation
  - Domain: Place

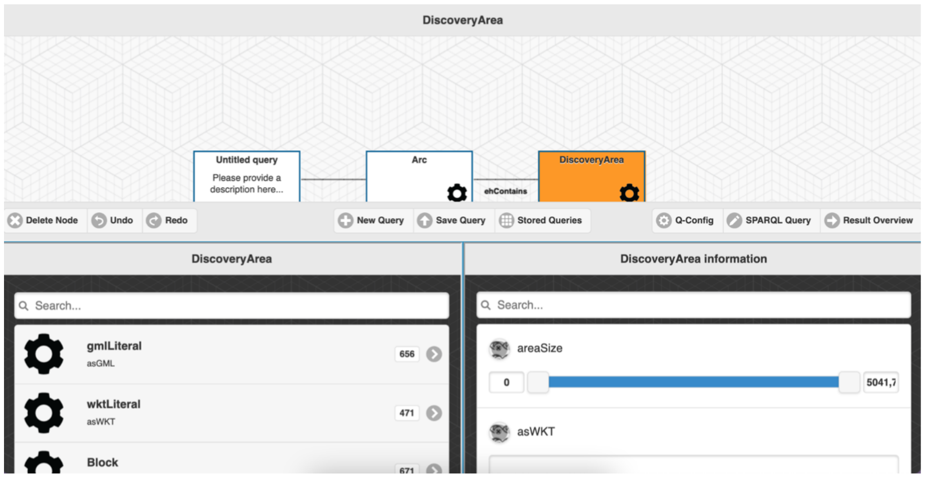


Figure 2.1: OptiqueVQS interface

2.4 Visual query interfaces and OptiqueVQS

A visual query interface is, as the name implies, a visual tool for creating queries. This can help users create complex queries without technical knowledge of the underlying query language.

OptiqueVQS is one such visual query system. It is driven by an underlying ontology, and uses SPARQL as a query language [15]. It has a visual interface where users can create queries. As we can see in figure 2.1, the interface has two main parts, one for building a query and one for viewing the query we're building.

On the top half of the interface we can see the query we're building. The default view is a graph, but we can also see the actual SPARQL query if we wish. Each node in the graph corresponds to one variable in the SPARQL query. We can focus on one node at a time. The background of a focus node turns orange, as we can see in figure 2.1. We can choose which node we wish to focus on at any given time by simply clicking on it.

While a node is in focus, the lower left pane shows pairs of object properties and classes we can extend the query with from that node. If we click on one of the options, we get an edge from our focus node to a newly added node, and the newly added node will become the focus node.

The lower right pane shows data properties we can filter on for our focus node. These filters allow us to make restrictions for the focus node, and what information we would like to see about the focus node in the results.

2.5 Related work

In this section we will look at previous work that is similar to or can be relevant for our work.

2.5.1 Reasonableness

The article ”Driving User Interfaces from FaCT” by Sean Bechhofer and Ian Horrocks describes a graphical interface where naive users can construct concept expressions without dealing with the underlying representation [1]. A mechanism called reasonableness is described for this purpose, which adds some restrictions when forming concepts. These restrictions are described by the paper as follows:

- Only “reasonable” concepts can be formed.
- Only a “reasonable” number of specialisation choices are offered at any point.

To ensure that only reasonable concepts can be formed, each concept has a list of concepts it can be reasonably joined with.

2.5.2 SPARQL Extension Ranking

A thesis called SPARQL Extension Ranking, written by Tom Fredrik Christoffersen, also focuses on OptiqueVQS [3]. The thesis focused on finding a way to rank possible options for what to extend the query with based on what is already selected, by identifying patterns in the query log.

Chapter 3

Problem analysis

Now that we have given a brief introduction to relevant concepts and explained the interface and basic usage of OptiqueVQS, we can go into a more detailed problem description. We will also describe five different information sources that can be used in a query system.

3.1 Detailed problem description

A limitation for the current version of OptiqueVQS is that information about subclasses is not presented in a useful way, we cannot navigate through a class hierarchy while creating a query.

When focusing on a class, we are presented with a list of classes that can be related to the focus object, along with the object property that relates them. Each object property is listed together with every possible class in its range, so one object property can be repeated many times if it has a large range. This means that the list can get very long and possibly hard to navigate, especially for very general classes.

Let's say we want to find humans who have a dog as a pet, and the dog's age is 2 years or younger. In the current solution we would do something like this:

- Find and choose Human in the list of classes.
- Scroll to (or search for) the property hasPet with the range Dog and select it.
- Restrict age under Dog information.

This looks simple, but it's not without problems. As mentioned above, the main problem is that a large number of options will make the list very long. This is an example of a list that can be long, as there can be many types of pets.

There are many possible solutions for adding subclass support, some possibilities will be presented in chapter 5 on page 17. When thinking of the possible solutions for subclass support, we must think of the usability and remember that the users of the query system might not have so much technical knowledge, so it should not be too complicated for them to use. Finding and adding a new property and node should be simple and preferably not require too many clicks.

3.2 Information sources

A query system can get its information from five different sources: data, ontology, admin input, user interaction and query logs. The information from these sources can be combined, and we will often want to use information from more than one of these sources at the same time.

One thing to think about, regardless of solution, is how options are shown, which options are shown and how many of them are shown. We are given some alternatives in the information sources described below.

3.2.1 Data

The data is the explicitly stated facts, and is an information source that is always needed. Without the data, the queries would not return any results. If a system is run using the data alone, it can only display options and relations that are present in the data, and it won't have any additional structural information.

If we want to sort the options by number of occurrences, or only want to show the most common classes or properties as options, we can also get this information from the data by looking at the number of occurrences in relevant triples. When building a query that includes humans who have a specific pet, we could use the data to find the most common pet types, as we would most likely be interested in a typical pet like Cat or Dog.

3.2.2 Ontology

While we need the data to show results, we don't necessarily need to use the data to show options in a query system. We can also use the information from an ontology for that. An ontology includes information about properties and classes, such as domain and range for properties and subclass information. The information from the ontology can help decide which options are valid and can be shown. If we know from the ontology that hasPet has domain Human, and Human and Cat are disjoint, hasPet will not be shown as a property option for Cat. If a system is run using the ontology alone, it will be able to present all valid options, but the queries will not return any results if there is no data.

By combining the data and the ontology we can get more information than the explicitly stated facts by using reasoning. If we know from the data that Anna has a pet, and we know from the ontology that the property hasPet has domain Human, we can infer that Anna is a human.

One thing to think about regarding ontologies is how top-down and bottom-up propagations are handled. A top-down propagation is when a subclass inherits a property from its superclass, and bottom-up propagation is the other way around. This can be sensible in query formulations; if filtering on fur colour for pets is sensible, then it is sensible for cats (downwards propagation). It is also sensible for animals, because some of those are pets (upwards propagation). But if these propagations go too many levels up or down, can we end up with all properties being available for all classes?

3.2.3 Admin input

Admin input is information that is provided by manual configuration. An example of this would be a sanctioning-like mechanism, as described in [1]. This has the advantage that we can configure information the exact way we want it, but the downside is that it can be a very time consuming job if there is a lot of information to configure, as it has to be done manually. It also has to be manually updated when the data or ontology changes.

3.2.4 User interaction

User interaction is the clicks and decisions made by the users while they are constructing a query. We can analyze those series of clicks and interactions and see if we find any patterns, which we in turn can use to make adjustments to our system. If we find a pattern where the users typically immediately do a delete or undo action after selecting a particular option, we could maybe avoid showing that option.

3.2.5 Query logs

Query logs contain information about previously made queries. These can be useful if we want to sort items by popularity or know which items are most likely to be selected, as can be seen in [3]. However, we do not always have a query log. If the query system is completely new or hasn't been in use for long, the query log will be empty or have very little information.

Chapter 4

Backend

In the current version of OptiqueVQS, the options shown on the frontend are decided by a navigation graph, which is created on the backend. In this chapter we will discuss some of the problems with the current navigation graph, and how we can replace it with a new and improved navigation approach.

4.1 The current ontology projection

In the article OptiqueVQS: a Visual Query System over Ontologies for Industry [15], the navigation graph for OptiqueVQS is defined as follows:

Definition. *Let O be an OWL 2 ontology. A navigation graph for O is a directed labelled multigraph G having as nodes unary predicates, constants or datatypes from O and s.t. each edge is labelled with a binary predicate from O. Each edge e is justified by one or more axioms [...].*

There are several criteria for when there should be an edge from a class A to a class B, or from a class A to a datatype, for instance:

- A SubClassOf: R restriction B, where restriction is one of the following: some, only, min, max or exactly
- A combination of range and domain axioms of the form: 'R Domain: A' and ‘R Range: B'

For class hierarchies, there are some propagation rules for applying edges. These propagation rules include both top-down propagation and bottom-up propagation:

- The rules for top-down propagation say that if there is an edge from a class A, that same edge should also be present for subclasses of A. For example, if a class Animal has an edge numberOfLegs, and Dog is a subclass of Animal, Dog would also get the edge numberOfLegs.

- The rules for bottom-up propagation say that if there is an edge from a class A, that same edge should also be present for superclasses of A. For example, if a class Dog has an edge hasOwner, and Animal is a superclass of Dog, Animal would also get the edge hasOwner.

The current ontology projection uses RDFox, in particular for propagation, and also for storing the navigation graph.

4.2 Critique of ontology projection

By only extracting basic edges using the OWL reasoner, the propagation is fast. If class hierarchies are not central, this is often good enough. The ontologies that were used with OptiqueVQS were often extracted from relational databases, which do not have a class hierarchy. These ontologies would then often have a flat data structure, so the handling of class hierarchies were not an issue.

If the rules for top-down and bottom-up propagation are taken literally, we could end up with every relation on every class. For example, let's say we have an edge hasDriversLicense from the class Human. With top-down propagation, this edge would be added to all subclasses of Human, including a potential subclass such as Minor, who should not be able to have a driver's license. With bottom-up propagation, hasDriversLicense would be added to superclasses of Human, such as Mammal. If this edge is propagated downwards again from Mammal to its subclasses, hasDriversLicense could be added as an edge from classes such as Dog or Elephant. To avoid this, some kind of ad-hoc restriction is needed for the propagation. Class hierarchies should therefore be handled differently than they are in the current solution.

Also, some axioms, such as $A \sqsubseteq \forall R.B$, are not a strong indication that there should be an edge in the graph, which results in the projection being more defined by what axioms are there syntactically than by some consistent semantic idea.

4.3 Replacing the navigation graph

To have more control over which options are shown, we needed to replace the navigation graph.

One thing in particular we wanted to have control over was which properties were shown as possible options for each class. Let's say we have a class Human, with the subclasses Adult and Minor, which are disjoint from each other, and an object property hasDriversLicense with Adult as domain. Here we want to show hasDriversLicense as a possible option for Human and Adult, but not for Minor.

To be able to do this, we went for an ontology driven approach: **unless the ontology entails that some combination of facts is impossible (e.g. x is a cat and x has a driver's license), it must be possible for the user to choose that combination (cat with driver's license).** For this to work, it is important that the ontology is expressive and has information about disjointness between classes, whether it's stated explicitly as a disjointness or as an implicit disjointness through reasoning.

This is different from data driven behavior, which only gives the user the possibility to choose things that occur in a specific data set. The number of possible choices in a data driven approach will depend on the data set, but there will typically be fewer choices than when using an ontology driven approach.

Let's show the difference between the two approaches with an example: if a data set don't contain any instances of humans that have a driver's license, it will not be shown as an option in a data driven approach. In an ontology driven approach it will be shown as an option (unless the ontology entails that it is impossible), even if the result will be empty when the query is run.

4.3.1 Our approach presented with Description Logic

The Description Logic (DL) concept $\exists R.\top$ describes all things that are R-related to something. A class C 'can' be R-related to something if the ontology O does *not* entail that C *cannot* be R-related to anything. I.e., given O and C, we need to find all R with $O \not\models (C \sqcap \exists R.\top \equiv \bot)$. This is actually a disjointness expression.

We have decided to make use of the functionality in DL reasoners (and the OWLAPI) to find disjoint classes in an ontology, if they are named classes. We will introduce a class Has_R for each relation R, and create an extended ontology O' from O by adding axioms $Has_R \equiv \exists R.\top$. The task then becomes to find for a given class C all the classes Has_R that are not disjoint with C.

4.4 Implementation

When we started working on developing our implementation, we created a small, separate program and a small example ontology that contained several disjointness axioms in order to be able to test our code easily.

The first thing we did was to load all classes, object properties and data properties into separate lists. To be able to use methods from the OWL API, we created an inner Java Class for each class where we kept a reference to their OWLClass.

After running a reasoner, we printed a list of all classes and their subclasses, a list of all the classes that were disjoint with each other, a list with the domains of all the object properties, and another list with the domains of all the data properties. After going through these lists and confirming that everything was correct, we could proceed to the next step.

By looking at code examples of how the OWL API could be used [10], we came up with an idea for how to find which properties should be available for each class. We started by looping through the list of object properties, and for each object property we used the OWL API to create a class expression for the class of individuals that have the current object property. We then used another method from the OWL API to find all the classes that were disjoint with that class expression. We then printed the name of each object property with a list of disjoint classes underneath each, so we could check if the results made sense, which they did.

Now that we knew which classes were disjoint with each object property, we needed to find a way to use that information to find out which object properties should be possible options for each class.

Because the disjoint method we used returned a set of classes that were disjoint with the object property, we realized we could use set operations to find the opposite; classes that were *not* disjoint with the object property. We created a new set and filled it with all the classes in the ontology. We then used the difference operation to remove all the disjoint classes from the newly created set, which left us with a set of all the classes that were not disjoint with the object property.

The final thing we had to do was to save this information for each of the classes in the set, so that it could be looked up later. We extended the inner Java Class to also have a list for possible object properties, and added the current object property to that list for all the classes that could have it. We then repeated this whole process for the data properties.

In the next two subsections we will first present an overview of the information we need to store and keep track of, and then we will present a step by step description of the algorithm we came up with.

4.4.1 Necessary information

We need to keep track of the following information:

- All object properties in the ontology
- All data properties in the ontology
- All classes in the ontology, including:
  - Reference to OWLClass
  - List of possible object properties for that class
  - List of possible data properties for that class
  - Number of superclasses
- Mapping between class URIs and the information stored for that class

4.4.2 Algorithm for finding possible properties

To find the possible object properties for all the classes, we did the following for each object property:

- Create a class expression for the class of individuals that have the current object property

- Find all classes that are disjoint with that class expression, and put them in a set disjointClasses
- Create a new set possibleClasses and fill it with all the classes in the ontology
- Remove all the classes in disjointClasses from possibleClasses
- For all classes in the ontology:
  - If possibleClasses contain the current class:
    Add object property to list of possible object properties for the current class

The process is the same for data properties.

These calculations are only done once, at initialization. The information found and stored can then be used as needed by looking up the URIs of the classes.

4.5 Efficiency

In Protégé, a program for editing ontologies, disjointness reasoning is usually disabled by default, because it is known to be slow. This can also be an issue for our implementation. In addition to disjointness reasoning being slow in itself, we are also looping through the entire list of classes one time each for every single property in the ontology. These two factors can result in a slow running time, especially for larger ontologies.

An important question for further development of this solution is therefore if there is a quicker way to do these calculations.

Chapter 5

Interface solutions

In this chapter we will present four different solutions for the interface. To decide which of these solutions to implement, we will do a user study, discuss the results and analyze the solutions, before coming to a conclusion on which solution to implement.

5.1 Presentation of possible solutions

We created four different solutions that we believe cover a good ground of possibilities in terms of how to select classes and properties with a class hierarchy in mind. We have one solution where we can select classes and object properties in pairs, two solutions where class and object properties are selected in two steps; one where object property is selected first, and one where class is selected first, and lastly a solution with three panes instead of the usual two; one pane for class selection, one for object property selection, and one for the data properties. These four solutions are described in detail below. The solutions are ordered by how similar they are to the current version of OptiqueVQS, with the first solution being the most similar and the fourth solution being the most different from the current version of OptiqueVQS.

5.1.1 Solution 1: Selecting class/object property pairs

In this solution we can select class/object property pairs in one step in the left pane. This is similar to the current version of OptiqueVQS, but instead of showing all possible class/object property pairs for each object property, only one pair is shown for each object property. The shown class is the one that is specified as the object property's range. After selecting a class/object property pair we can refine the class in the right pane. To make the initial class selection consistent with the rest of the solution, only the top level classes are shown in the left pane. These can then be refined in the right pane after selection.

Wireframes of solution 1 can be seen in figure 5.1 and figure 5.2 on page 19.

Example interaction

If we were to build a query to find all humans who have a pet that is a dog, the interaction steps would be as follows:

1. Left pane: Select the class Human.
2. Left pane: Select the object property/class pair hasPet Human.
3. Right pane: Select the subclass Dog under ”Refine class”.

5.1.2 Solution 2: Class and object property selection in two steps - object property first

We start by selecting an initial class through a class hierarchy in the left pane. After selecting this class, we can expand the query by doing a class and object property selection in two steps. First a list of possible object properties is presented. After an object property has been selected, we can navigate a class hierarchy of possible ranges to find the desired class.

Wireframes of solution 2 can be seen in figure 5.3 and figure 5.4 on page 20.

Example interaction

If we were to build a query to find all humans who have a pet that is a dog, the interaction steps would be as follows:

1. Left pane: Select the class Human.
2. Left pane: Select the object property hasPet.
3. Left pane: Expand the class Animal.
4. Left pane: Select the subclass Dog.

5.1.3 Solution 3: Class and object property selection in two steps - class first

We start by selecting an initial class through a class hierarchy in the left pane. After selecting this class, we can expand the query by doing a class and object property selection in two steps. First a list of classes that can be related to the chosen class is presented in a class hierarchy. After a class is selected, a list of possible object properties is shown, and the user can select the property they want.

Wireframes of solution 3 can be seen in figure 5.5 and figure 5.6 on page 22.

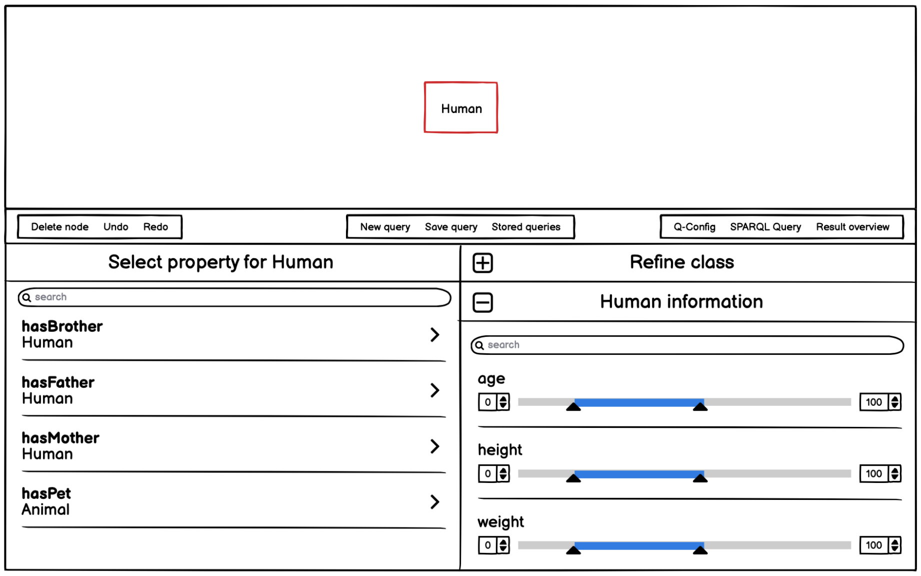


Figure 5.1: Solution 1 - Selecting class/object property pairs in the left pane.

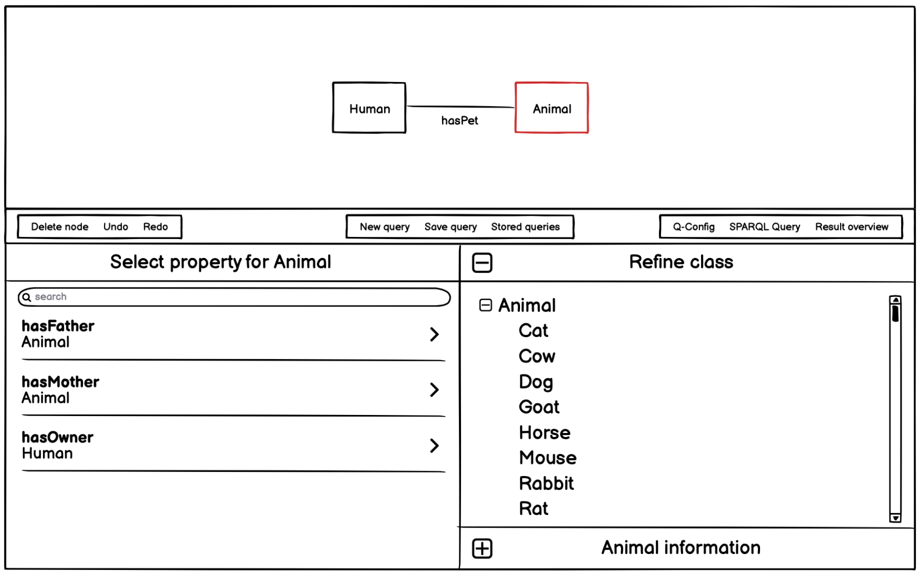


Figure 5.2: Solution 1 - The right pane gives the possibility to refine the chosen class.

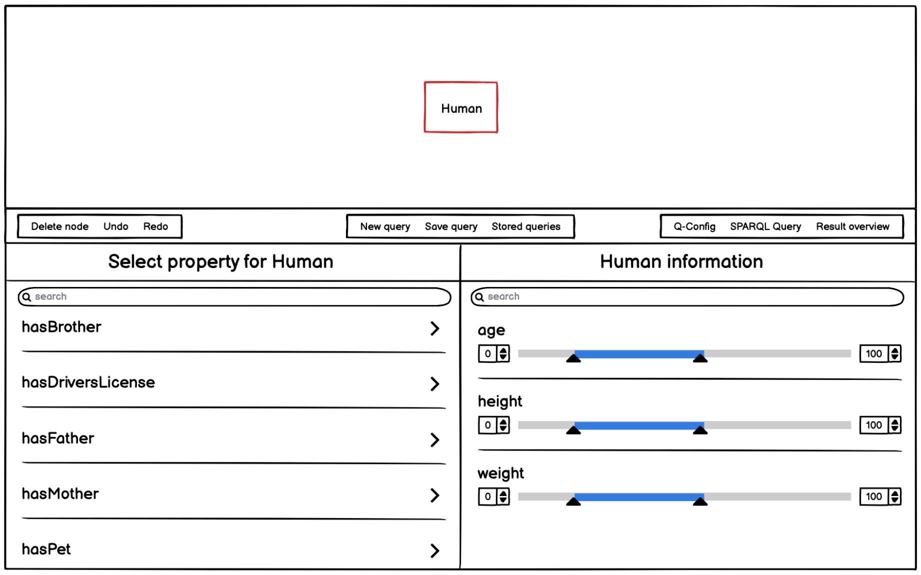


Figure 5.3: Solution 2 - To add a new node, we have to start by choosing an object property.

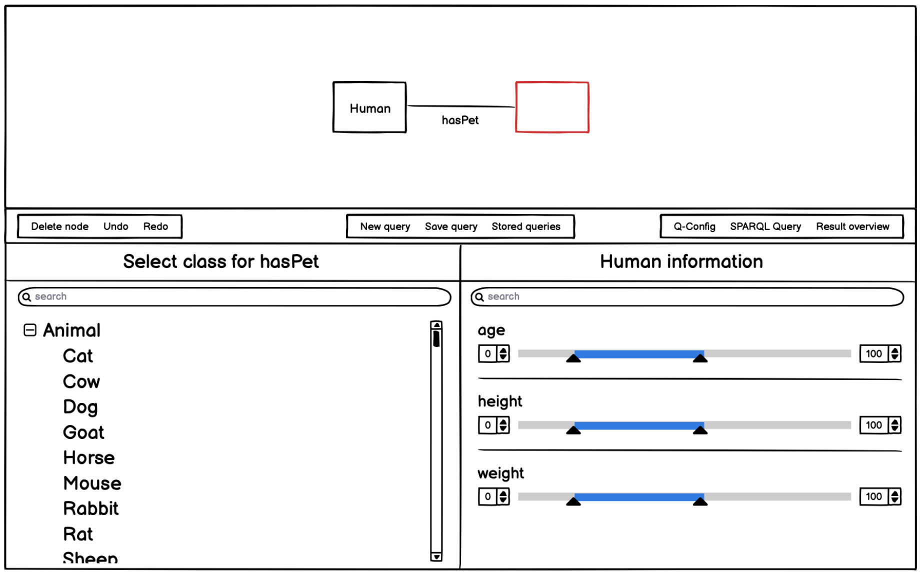


Figure 5.4: Solution 2 - After an object property has been selected, we have to choose a class.

Example interaction

If we were to build a query to find all humans who have a pet that is a dog, the interaction steps would be as follows:

1. Left pane: Select the class Human.
2. Left pane: Expand the class Animal.
3. Left pane: Select the class Dog.
4. Left pane: Select the object property hasPet.

5.1.4 Solution 4: Three panes

In this solution we have separate panes for object property and class selection, which gives us three panes instead of the usual two; one for object property selection, one for class selection, and one for restricting the current class. After selecting the initial class from a class hierarchy, the left pane is filled with a list of possible object properties, and the middle pane is filled with a list of classes that can be related to the chosen class. As we navigate the class hierarchy in the middle pane, the list of possible properties changes, and only those with a range within the selected class is shown. If we start by selecting a property, the list of classes will change and only the classes with the correct range are shown.

Wireframes of solution 4 can be seen in figure 5.7 and figure 5.8 on page 23, and figure 5.9 on page 24.

Example interaction

If we were to build a query to find all humans who have a pet that is a dog, the interaction steps would be as follows:

1. Middle pane: Select the class Human.
2. Left pane: Select the object property hasPet.
3. Middle pane: Expand the class Animal.
4. Middle pane: Select the subclass Dog.

Or alternatively:

1. Middle pane: Select the class Human.
2. Middle pane: Expand the class Animal.
3. Middle pane: Select the subclass Dog.
4. Left pane: Select the object property hasPet.

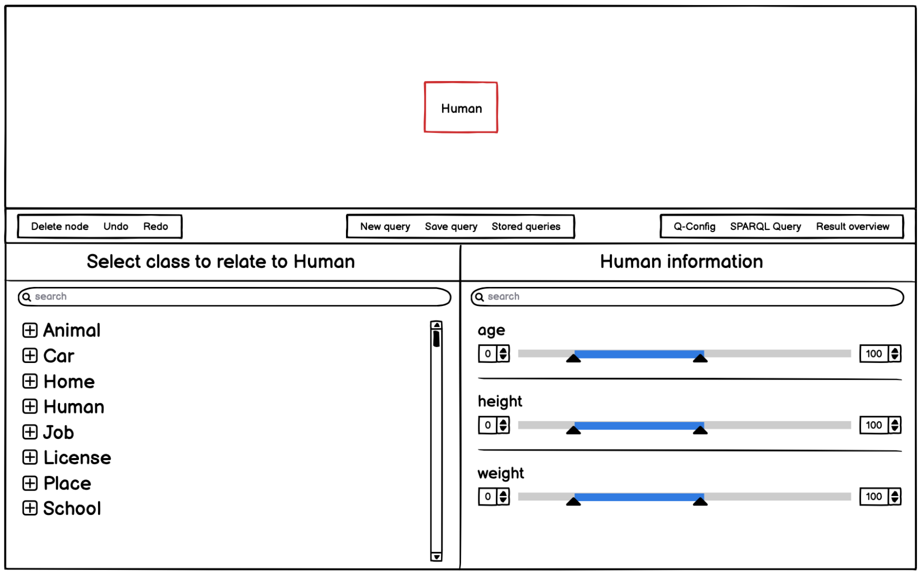


Figure 5.5: Solution 3 - To add another node, we start by selecting a class to relate to the current class.

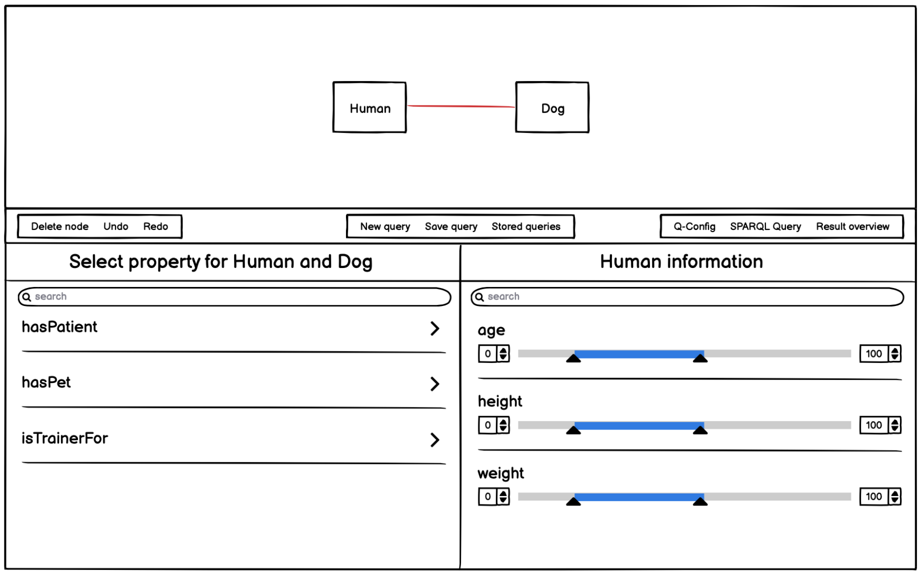


Figure 5.6: Solution 3 - After selecting a class, we have to choose one of the available properties.

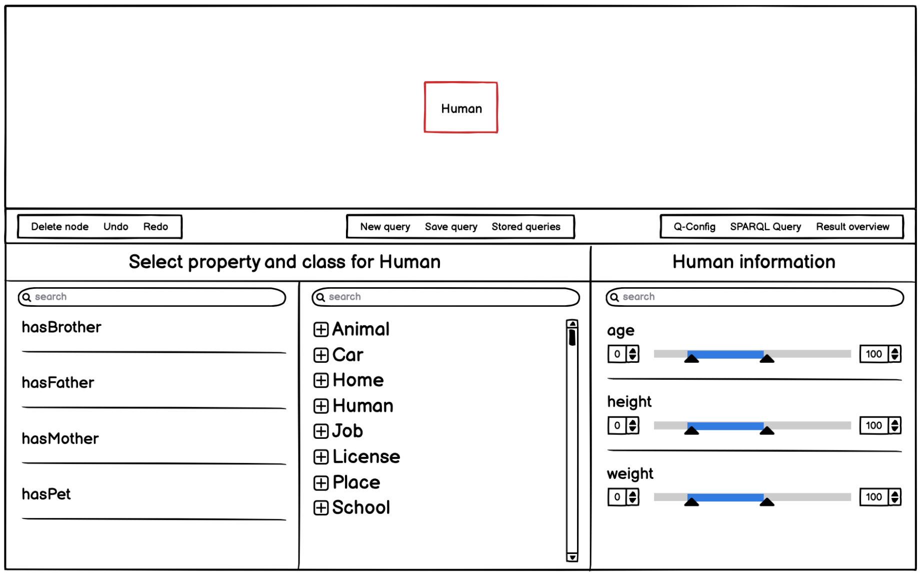


Figure 5.7: Solution 4 - Separate panes for object property and class selection.

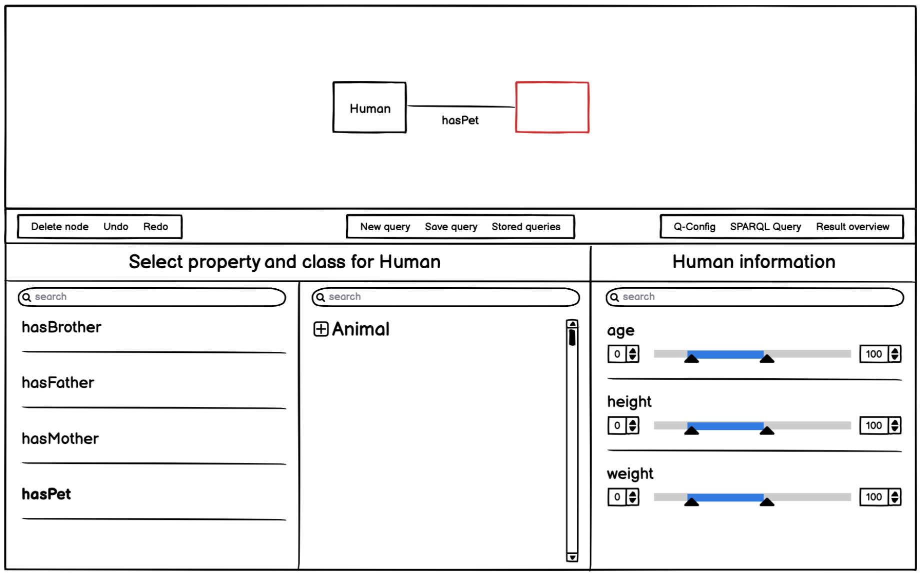


Figure 5.8: Solution 4 - If an object property is selected, the list of possible classes changes.

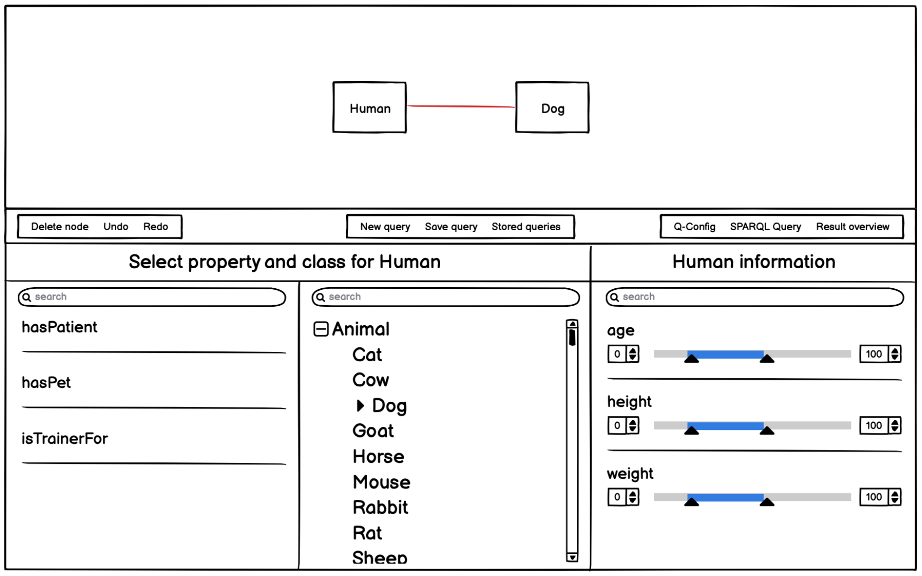


Figure 5.9: Solution 4 - If a class is selected, the list of possible object properties changes.

5.2 User study

To be able to make an informed choice for which solution(s) to implement, we decided to do a user study where the participants could click through interactive prototypes of the four proposed solutions.

5.2.1 Study setting

For the user study, we decided to use the thinking aloud method. This is a simple and effective method where the users are asked to continuously think aloud as they are clicking through the interface and performing the tasks they are given. They are asked to express any assumptions they have, expectations for what will happen, and any difficulties they encounter. This method lets us discover what the users think, and if they have any misconceptions or difficulties, it is usually a sign that something in the interface needs to be changed. [7]

To prepare for the user study, clickable wireframes were made for each of the four proposed solutions. The user study was done via video meetings, with one participant at a time. We had one person guiding the study by telling the participants the task they had to perform and ask questions at the end, and another observing and taking notes. The participants were given a link to the wireframes, and were then asked to share their screen with us. The task they were given, and had to do for each of the four solutions, is presented in section 5.2.3 on the following page.

5.2.2 Participants

We had three participants for the user study. A short description about each participant can be seen in table 5.1. While these participants not necessarily are the in the main target group for OptiqueVQS, they can still help us provide insight and valuable feedback for what they think of the different solutions.

Participant	**Relevant information about the participant**
P1	P1 is a researcher and implementer, and is often focused on error finding.
P2	P2 is a software developer. He has not been involved with the development of OptiqueVQS, nor does he have any experience with using OptiqueVQS.
P3	P3 has not been involved in the development of OptiqueVQS, but has been demoing it to the oil and gas industry, and is the most similar to an ordinary user out of the four participants.

Table 5.1: Profile of the participants in the study

5.2.3 Task

The participants were given one task they had to do, which was to build the following query:

- Find all humans who have a pet that is a dog.

The participants had to build this query in each of the four solutions. We chose a task that was simple, but still included a subclass selection, as subclass selection was the central part of our solutions.

5.2.4 Feedback

Feedback for solution 1

All three participants went through the first solution fairly quickly and easily, and it was generally seen as easy to use. During one of the interviews it was mentioned that it made sense to add new nodes on the left, and adjust the current one on the right. Another participant mentioned that he got used to looking at the left pane, and therefore it was strange/irritating to have to look somewhere else (right pane) to finish the query.

Feedback for solution 2

The second solution was also gone through fairly quickly and easily by the participants, and one of the participants said that this solution made sense graphically. Another participant said it felt like more clicks than the first solution, but thought it was positive that the entire query could be made by only focusing on the left pane.

Feedback for solution 3

The third solution was the solution that took the longest to figure out for all three participants. After selecting the initial class, there was a pause where they didn't know what to do next. They eventually figured it out after being told to look at the heading, which had changed from telling them "select a class" to "select a class to relate to (chosen class)". None of them noticed the heading change by themselves. They felt that this solution was confusing, and not very logical or convenient. It was also mentioned that they want a flow from left to right, and this solution goes against that flow.

Feedback for solution 4

The fourth solution was overall seen as an interesting solution, and one of the participants said this was their favorite solution of the four. Being able to see both object properties and classes at the same time gives the user more freedom to explore, and they can decide what they want to start with. One complaint was that the flow went against the reading direction, because the initial class selection happens in the middle pane while the two other panes are blank, and they wished we would start on the left and not in the center.

5.3 Discussion and analysis

5.3.1 Solution 1

Number of clicks

Minimum number of clicks for example task: 3

Discussion

As mentioned in section 5.1.1 on page 17, we made the choice to only show top level classes for the initial class selection, with the ability to refine the chosen class in the right pane. The main reason for doing this was to make this part consistent with the rest of the solution, where we also only show one option for each object property. By only showing one class/object property pair for each object property, the object property selection list should become much shorter and easier to navigate. In the right pane we can add restrictions to the selected class, so it makes sense to also refine the classes there.

Whether or not it is a good idea to only show top level classes in the left pane can depend on both the structure and size of the ontology, as well as the users familiarity with the ontology. If the ontology is very large and only has a few top level classes that all are very general, it can be difficult to know under which class to find the subclass we're looking for. Likewise, if the user is unfamiliar with the ontology, it might be hard to know how to get to a specific subclass, and the user might prefer to look through the class hierarchy before making a selection.

Something that was talked about during the user studies regarding the first solution was that it is the only solution where we don't have an "in between state" where we can't run the query. In this solution we always have a "complete" query after clicking on something. In the other solutions we will at some points have a partial query where either the object property or the range class is empty, which would most likely have resulted in an error if the query had been run. This could have been solved by having "next step" and "add to query" buttons, but that would again have resulted in needing more clicks.

5.3.2 Solution 2

Number of clicks

Minimum number of clicks for example task: 4

Discussion

In this solution we are "guided through" the process of adding a new node, where we first are asked to select an object property and then a class. All the steps are done in the left pane, so the users don't have to shift their focus to other parts of the screen.

A downside to only showing the object properties first is that it can result in users having to go back and forth between the object property and class selections if they are just looking at the available options or didn't find the expected class after selecting an object property. Whether this will be a problem will depend on the object properties in the ontology. With specific object properties such as hasMother or hasPet we will know which classes to expect, but it can be harder for more general object properties like "owns".

5.3.3 Solution 3

Number of clicks

Minimum number of clicks for example task: 4

Discussion

As with solution 2, here we are also "guided through" the process of adding a new node, the difference being that in this solution we first select a class, and then a property.

The idea behind this solution was to decide which two classes to connect before selecting the object property that connects them. Selecting both classes first means that the list of possible object properties will be much shorter when it is presented to us, as only those that can be used to connect the two classes will be shown.

While this seemed like a good idea, it did not work so well in practice. As described in section 5.2.4 on page 25, all three participants in the user study were confused by this solution.

5.3.4 Solution 4

Number of clicks

Minimum number of clicks for example task: 4

Discussion

This solution gives the best overview of the available options to extend the query with, as we can see both properties and classes at the same time, and these will change depending on what is selected. All information is shown on one screen, there is no need to go back and forth to change view between properties and classes.

This solution also gives the users the most freedom to explore the ontology, and lets the users choose which order they want to do things in when adding new nodes; either select an object property first and then a class, or select a class first and then an object property. This is the only solution which gives the users that choice, in the other solutions these steps must be done in a set order.

This also means that we aren't guided through the selection in the same way we were in solution 2 and 3. Seeing all these selection widgets at the same time might be overwhelming for some users, making it harder for them to know where to focus.

5.3.5 General thoughts

A general observation when the participants interacted with a hierarchical structure: the participants mostly clicked on the plus beside the class to expand that class and view its subclasses, and directly on the name to select that class. This was also the intention when making the interactive wireframes, so it was good to get this confirmed.

5.4 Deciding on a solution to implement

When it was time to decide which of the four solutions to implement, solution 3 was the first to be discarded, because of the difficulties encountered during the user studies. For the three remaining solutions, solution 1 and 2 were pretty similar in terms of functionality and usability, so we wanted to eliminate one of them. We decided to discard solution 2 and keep solution 1, because it required less clicks and was a bit more expressive than solution 2.

The two remaining solutions were then solution 1 and solution 4. If we had had the time to implement two solutions, we would have implemented both solution 1 and solution 4, and done a usability study where we compared the usability between the two solutions. Solution 1 is the most similar to the current version of OptiqueVQS, and solution 4 is the most different from the current version of OptiqueVQS, so a comparison between the two would have been interesting to do.

Because of limited time, we could only implement one solution, so we had to choose between solution 1 and solution 4.

Since we only could choose one solution, we felt that it made the most sense to go for solution 1. Solution 1 is the most similar to the current version of OptiqueVQS, but we still believe it is an improvement over the current version. Its similarity will make sure that it won't be too big of a change for existing users. In the user study, it was generally seen as easy to use, and it was the solution that required the least amount of clicks to create the query for the example task. It was also the only solution out of the four where we always have a complete query, no matter what we do.

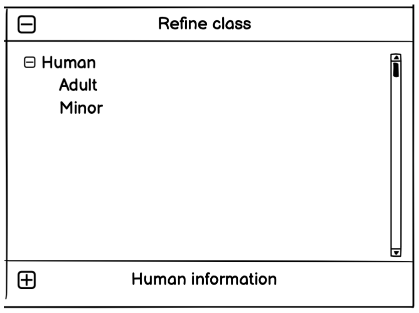
Chapter 6

Frontend implementation

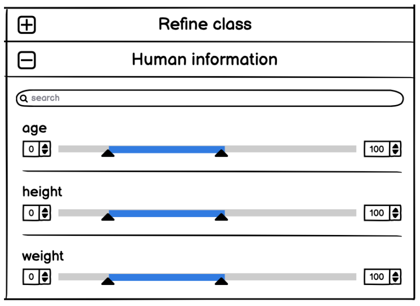
In this chapter we will describe the process of implementing the interface solution we chose in the previous chapter, including choices that were made along the way, and difficulties we encountered.

6.1 Rethinking the wireframes

In the wireframes for solution 1 we had originally designed the right pane to have two collapsible parts, a new part for refining the chosen class with a class hierarchy navigation, and the existing part where we can filter on data properties. The idea was to only show one of these two parts at a time, which gives both parts a good amount of space on the screen when they are open. Sketches of this can be seen in figure 6.1.



(a) Refine class



(b) Class information

Figure 6.1: The right pane with two collapsible parts

After thinking more closely about it, we believed that having two collapsible parts could result in a more cluttered interface that required more clicks and could possibly cause more confusion for the users. Switching between the two parts will require extra clicks, and some users might not notice that it is possible to switch between the two parts, and therefore miss out on an important part of the filtering functionality.

With this in mind, we decided that we would rather have all the filtering functionality in one pane. The original version of OptiqueVQS already had a small part at the bottom of the right pane for selecting direct subclasses, so we decided to use that as a basis, and work on modifying it.

The first thing we did was to move this subclass selection part from the bottom of the pane to the top, because it is an important part of the query building in this new solution, so it should be easily available. We also added a heading "Refine selection" to make it more visible.

6.2 Class hierarchy navigation

The next thing to work on was how to present the subclass selection. Because the subclass selection no longer had an entire pane to itself, we didn't want it to take up too much space. It was therefore decided to not go for a typical hierarchical navigation tree where the users can expand/collapse subclasses, the main reason being that it would take up too much space.

6.2.1 First implementation

We decided to try a step-by-step solution, where we only show a list consisting of one level of subclasses at a time. When the user clicks on a subclass, that class becomes the main class and its direct subclasses are now shown. Above the subclass selection we wanted a breadcrumb line to keep track of the selections made, and show where in the class hierarchy we are now. The elements in the breadcrumb should be clickable, and take us back to the part in the class hierarchy that was clicked. An up-close look at the type of breadcrumb we envisioned can be seen in figure 6.7 on page 35.

There shouldn't be a big difference in the amount of clicks or usability between this solution and a regular hierarchical navigation. In a regular hierarchical navigation we would have to expand and collapse subclasses, while with this solution the immediate subclasses are already expanded, and the next level of subclasses will be shown as soon as a subclass option is clicked. If we at some point want to go back, we can use the breadcrumb to navigate back to a point higher up in the class hierarchy. One scenario where users might prefer a regular hierarchical navigation instead of this solution is if they are unfamiliar with the ontology, and want to look through the class hierarchy without having to "make a commitment" by selecting a subclass.

After implementing a prototype of this solution, we saw that even one level of subclasses could take up a lot of space on the screen. This can be seen in figure 6.2 and figure 6.3 on page 33. If a user wants to filter on a data property, they will have to scroll past a (possibly long) list of subclasses to find what they're looking for. If the subclass selection covers the entire right pane, the user might not even know that there are more options below.

We also had some difficulties with getting the subclass selection part to communicate with the rest of the system. It worked fine when we tested it with manually added classes, but when trying to run it with the classes from the ontology, we were not able to update the class list after a selection had been made. These difficulties, along with the fact that this implementation still took up a lot screen space without any way to hide it, made us look at other possibilities for implementing the subclass selection.

6.2.2 Second implementation

To get back to a working solution, we went back to the existing subclass selection we used as a basis. This existing solution does not immediately show any subclass options, it only has a title "Type" and a button with the text "Please select". When the button is clicked, a pop-up window with a list of the available subclasses is shown in the right pane. This pop-up window can be seen in figure 6.4 and 6.5 on page 34.

When thinking about how even our intended solution ended up taking up a lot of space, having the subclass selection in a pop-up window seemed like a good idea that would solve the issue of the subclass selection taking up too much space. This gives the users the ability to see that there are other options than subclass selection below, and if they wish to select a subclass, they can click on the button to open the selection window. This does add two extra clicks (one for opening the window for subclass selection and one for closing it), but we still believe it improves the overall usability.

This existing solution for subclass selection only showed the direct subclasses of the chosen class, and didn't have any way of going further down in the class hierarchy, so we still needed to make some improvements to this solution. We still wanted to use the step-by-step method for subclass selection that we described at the start of section 6.2.1 on the preceding page, including the breadcrumb.

We started by adding the ability to show the next level of subclasses after an initial subclass selection had been made. After a subclass selection had been made, we had to send a new request to the backend to retrieve the subclasses of that selected class. To show these new subclasses, we deleted the part of the HTML-code that showed the previous subclasses and added it back again, this time with the new list of subclasses as options.

For the breadcrumb, we kept a list of the subclass selections that had been made. Every time a subclass was selected, it was added to the list, and the breadcrumb was updated to also show the new selection. We can see how this second solution ended up looking, including a close-up look at the breadcrumb, in figure 6.6 and 6.7 on page 35.

6.2.3 Implementation issues

When a class in the breadcrumb was clicked, the idea was to remove all entries in the list with a higher index than the class that had been clicked, and go back

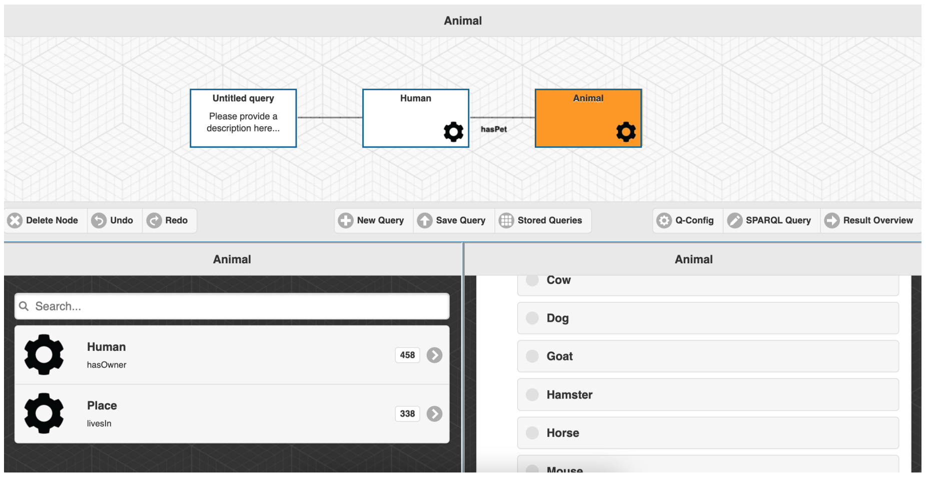


Figure 6.2: First implementation - long list of subclasses

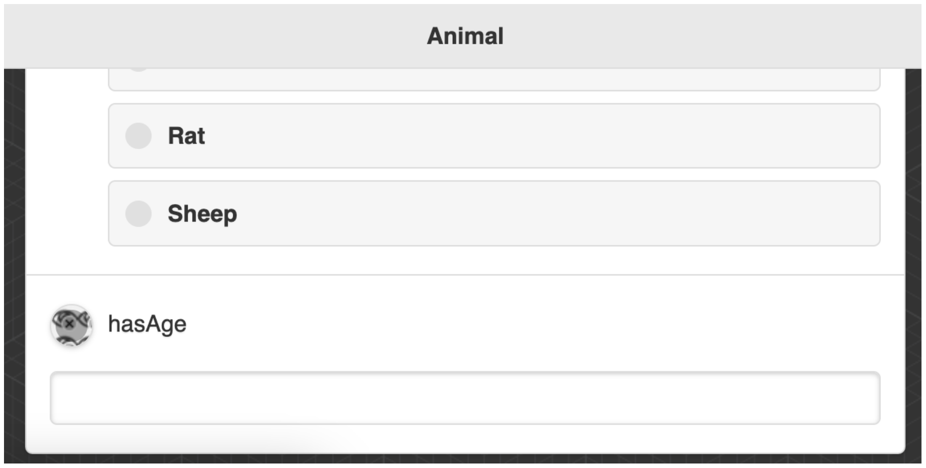


Figure 6.3: First implementation - After scrolling to the bottom of the subclass selection, we can see the data property options

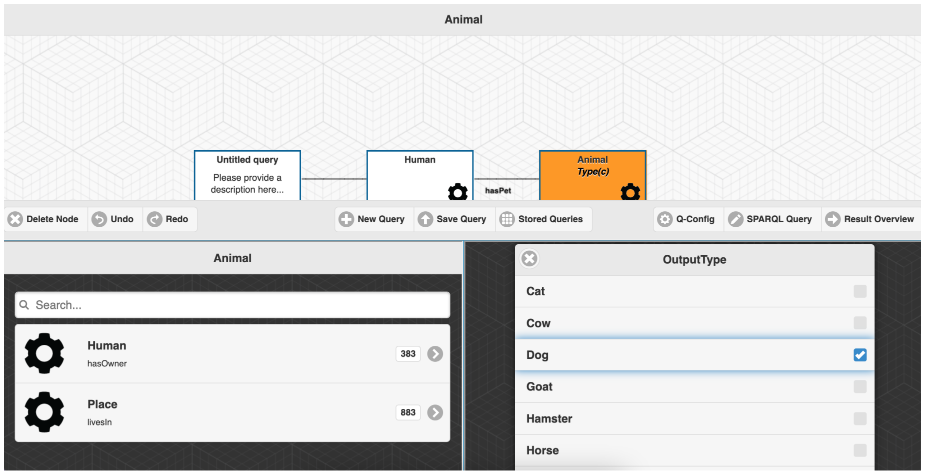


Figure 6.4: Pop-up subclass selection window

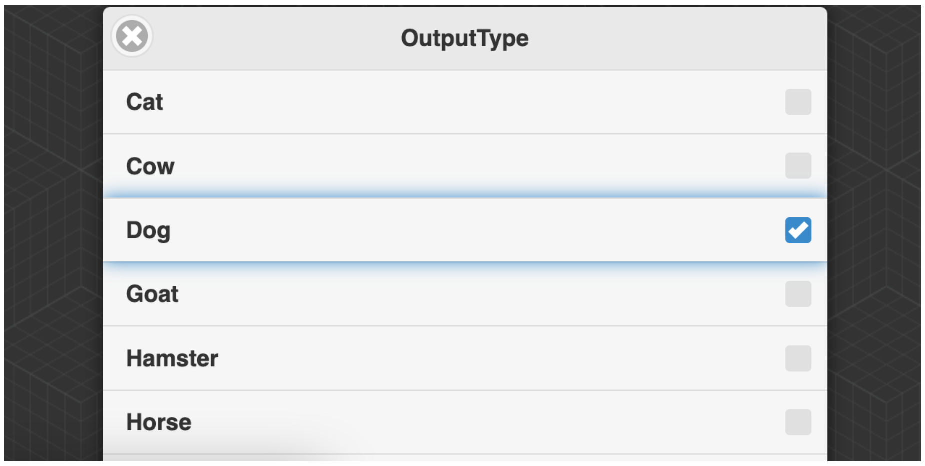


Figure 6.5: Closer look at the pop-up subclass selection window

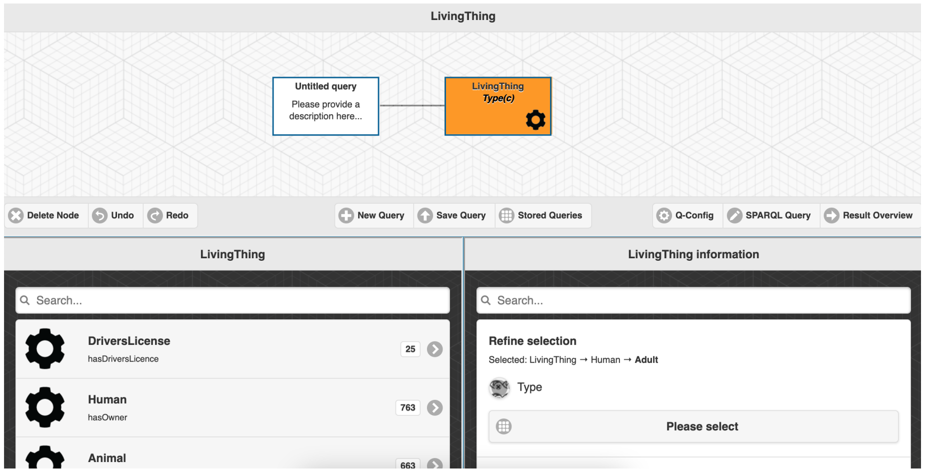


Figure 6.6: Overview of the second implementation

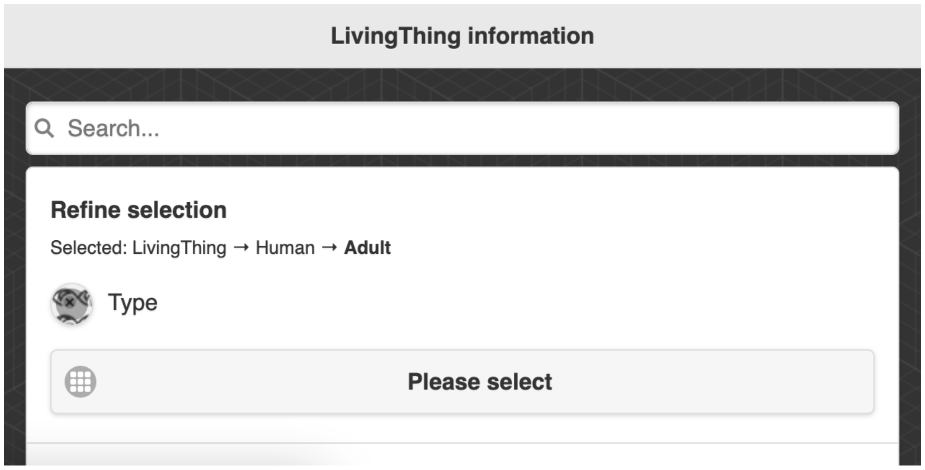


Figure 6.7: Close-up look at the second implementation

to showing the subclasses of the class that had been clicked. This worked fine for the prototype in section 6.2.1, where we used manually entered testing data. However, when we tried to use it with the data we retrieved from the ontology, we encountered the same issue as in section 6.2.1, where we were not able to call the function that updated the data. We believe that the cause of this could be that all the other parts of the interface were built using components from jQuery Mobile, while the breadcrumb functionality was created with "regular" HTML and JavaScript.

With these issues, we could no longer use the breadcrumb to navigate back to a point further up the class hierarchy, which limited the functionality and flexibility of our intended solution. Because of limited time, we did not have the time to come up with and implement a whole new solution for subclass selection, so we decided to keep working with this one. As long as we were able to select multiple levels of subclasses, we kept the breadcrumb as a visual guide to keep track of where we were in the class hierarchy.

After some testing, we discovered that the subclass selection did not always work properly either. When we added a new node and tried to refine the class, everything worked as intended. We could select subclasses that were multiple levels down in the class hierarchy, and the selected subclasses were also reflected correctly in the generated SPARQL query.

The problem arose when we added another new node, and then went back to the previous node where we had made a subclass selection. The part for subclass selection now showed the initial class and its subclass selection, and it was not possible to click on the button to make another selection. The previously selected subclasses were still shown in the SPARQL query, so the selection had been saved, but it was not otherwise reflected in the interface.

6.2.4 Final implementation

With the limited time we had left, we were not able to find a way to fix the issues we encountered and described in section 6.2.3 on page 32, and we had already scheduled a usability study for our new version of OptiqueVQS. Therefore, we had to choose between presenting a version that supported multiple levels of subclass selection, but didn't work 100% of the time, or go back to a fully working version that only supported one level of subclass selection.

We decided that it was more important for us to be able to present a fully working version, so we removed the breadcrumb and the functionality for selecting multiple levels of subclasses. If we designed our example ontology to only have one level of subclasses, we could still use this version to have a usability study with subclasses as a central part. The final version of our implementation can be seen in figure 6.8 on the next page.

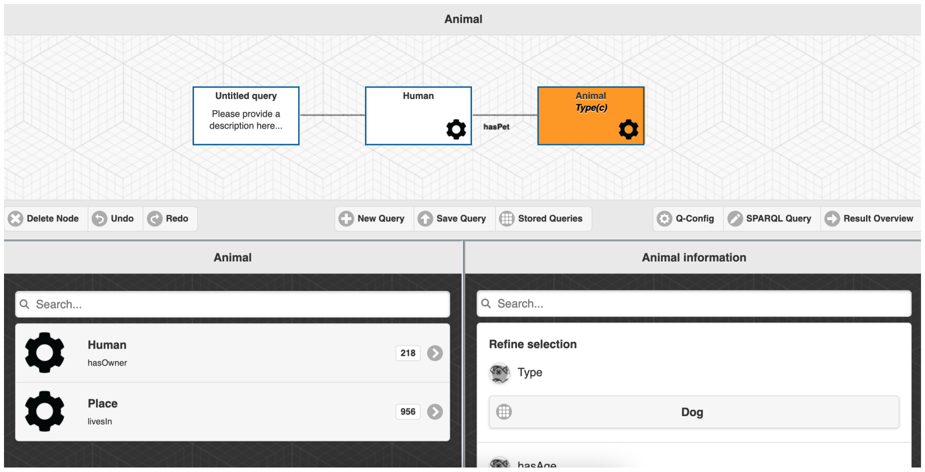


Figure 6.8: Final implementation

Chapter 7

Evaluation

After making changes to both the backend and the frontend, we want to evaluate this new version of OptiqueVQS by doing a usability study. We will start by describing the setting for the study, give some brief information about the participants, and explain the tasks used for the study. We will then present the feedback from the study, before having a discussion around our findings.

7.1 Usability study

7.1.1 Study setting

As with the user study in section 5.2.1 on page 24, this usability study was also performed as a think aloud study, where the participants are asked to think aloud as they perform the tasks they are given. For this study, the participants were given three tasks they had to perform while we observed them. These three tasks, along with our reasoning for choosing those tasks, are presented in section 7.1.3 on the next page.

Because our new version of OptiqueVQS only was installed on one computer, it was easiest to do this user study in person. To make it easy for us to be able to see what the participants were doing without having to stand directly behind them, the computer screen was mirrored to a larger monitor. We did the study with one participant at a time, and in total we had four participants, which are described in section 7.1.2. After the participants had completed the tasks, they were asked to fill out a System Usability Scale (SUS), which is presented in section 7.1.4 on page 40.

7.1.2 Profile of participants

We had four people participating in the user study. An overview of these four participants can be seen in table 7.1 on the following page.

Participant	**Relevant information about the participant**
P1	P1 is a researcher and implementer, and is often focused on error finding.
P2	P2 is a software developer. He has not been involved with the development of OptiqueVQS, nor does he have any experience with using OptiqueVQS.
P3	P3 has worked on the backend of OptiqueVQS for his PhD, and therefore has a lot of knowledge of OptiqueVQS.
P4	P4 has not been involved in the development of OptiqueVQS, but has been demoing it to the oil and gas industry, and is the most similar to an ordinary user out of the four participants.

Table 7.1: Profile of the participants in the usability study

7.1.3 Tasks

Three tasks were created for the experiment:

1. Find all cats.
2. Find all dogs who have an owner.
3. Find all humans who live in a city and have a pet that is a dog.

The ontology being used for these tasks is the same ontology that was introduced as an example ontology in section 2.3.3 on page 5. The tasks are designed to test important and relevant parts of the system, such as object property selection and subclass restriction. To give the participants a chance to get familiar with the system, we start with a simple task, and then the tasks become increasingly more complex.

In the first task, participants will have to select a superclass (Animal) and figure out how to change it to the desired subclass (Cat). The goal for this task is to understand that we can select a general class in the left pane, and then change it to a more specific subclass in the right pane. This is important for the participants to learn, as restricting subclasses is a central part of what we have been working on and want to test in this new version of OptiqueVQS.

In the second task, the participants have to do a similar subclass restriction as in task 1, and then add an object property (hasOwner). We didn't specify a range for the object property, because the focus in this task is on understanding how to expand the query by adding new edges, so we just wanted the participants to add the correct edge without thinking about anything else.

After the first two tasks, the goal is for the participants to have a basic understanding of how OptiqueVQS works. The third task is therefore a more complex query, where the participants have to add *two* edges from one node, and add subclass restrictions to both.

7.1.4 Results

Observations and feedback

The first participant (P1) spent the most time on task 1. After selecting the initial class (Animal), the participant seemed very focused on the left pane, and it took some time to figure out where to restrict the type from Animal to Cat. When he didn't find anything on the left pane, he shifted his focus to the right pane, and found the type restriction. After completing the first query, he didn't have any problems constructing the last two.

The second participant (P2) had not used OptiqueVQS before, and got a quick introduction before attempting to do the tasks. He finished the first task quickly and without any problems. On task 2, he paused after doing the subclass restriction, and it took some time to figure out how to add the object property (hasOwner). After figuring it out, he said that he did not see the object property label at first; it was too small, and he only focused on the bold text above. For task 3, he started by correctly selecting the first part of the query (Human hasPet Dog). For the second part of the query, he continued from Dog and added the edge livesIn City, which resulted in the entire query being in a straight line. After a short while, he figured out that he could go back to a previous node, and remade the query with two outgoing edges from Human.

The third participant (P3) started task 1 by trying to search for Cat in the left pane, which did not return any results. He then took a look at the options, selected Animal, and changed it to Cat in the right pane. Task 2 was completed quickly, and the participant said the task was easy after doing task 1. The participant did not have any problems doing task 3, and finished that task quickly as well. For task 3 he mentioned that it was important to know that we can click on any node in the graph to change focus.

The fourth participant (P4) was familiar with using OptiqueVQS, and went through all three tasks fairly quickly and without encountering any problems. While performing the tasks he said that he understood the ontology, and that it is important for the users to know/understand the ontology in order to be able to use the system.

System Usability Scale (SUS)

After completing the tasks, the participants were asked to fill out a System Usability Scale (SUS). The SUS is a widely used questionnaire that has been reported to account for as much as 43% post-test questionnaire usage [14]. The SUS consists of 10 statements about usability, and the participants have to rate each of these statements on a scale from 1 (strongly disagree) to 5 (strongly agree). These statements and their answers can be seen in table 7.2 on the following page.

To calculate the SUS score, the first step is to find the score contribution for each statement. For statements 1, 3, 5, 7 and 9, the score contribution is the answer minus 1. For statements 2, 4, 6, 8 and 10, the score contribution is 5 minus the answer. The next step is to find the sum of the ten score contributions, and the final step to find the SUS score is to multiply the sum by 2,5. [2]

The SUS scores can have a range from 0 to 100. The scores for this experiment ranged from 75 to 85, with an average of 81,25, as can be seen in table 7.3.

Statement	**P1**	**P2**	**P3**	**P4**
1. I think that I would like to use this system frequently	5	4	2	5
2. I found the system unnecessarily complex	2	1	1	2
3. I thought the system was easy to use	4	4	5	4
4. I think that I would need the support of a technical person to be able to use this system	1	2	1	2
5. I found the various functions in this system were well integrated	3	5	4	5
6. I thought there was too much inconsistency in this system	2	2	1	1
7. I would imagine that most people would learn to use this system very quickly	4	4	5	5
8. I found the system very cumbersome to use	4	2	3	1
9. I felt very confident using the system	4	4	5	5
10. I needed to learn a lot of things before I could get going with this system	1	2	1	4

Table 7.2: SUS statements and results

Participant	**SUS Score**
P1	75
P2	80
P3	85
P4	85
Average score	81,25

Table 7.3: SUS scores

Other feedback

In this section we will describe some feedback that is not directly related to our solution and what we implemented, but can still be relevant for the usability related to query building and subclass selection.

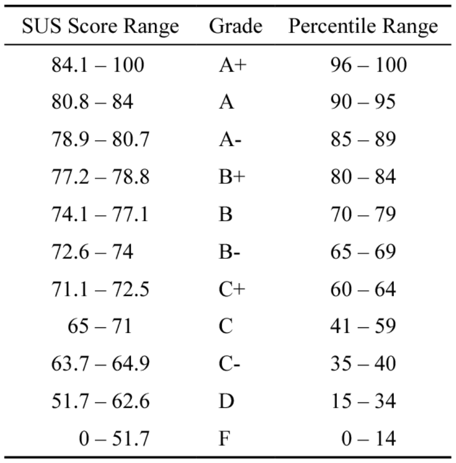
The left pane shows pairs of classes and object properties for adding new nodes. The class name is displayed in bold text, and the object property is displayed in regular text underneath. All four participants expressed that they wished the object property was in focus instead of the class.

When selecting a subclass, a pop-up window with the available options are opened. This window has an X in the top right corner, which can be used to close the window after a selection has been made. Three out of four participants were concerned about how to exit this window without discarding the selection they had made. It was mentioned that X is associated with cancel, and they wished there was an OK button or something similar.

After selecting a subclass, the node in the graph view is updated to show that a subclass has been selected. This is shown by putting the text "Type(c)", under the class name. We can see an example of what this looks like in figure 6.8 on page 37. One of the participants said it would have been better if they could have seen the name of the selected subclass here, instead of just "Type(c)".

7.2 Discussion

Using data from the entire set of 446 surveys, the average SUS score has been found to be 68. A curved grading scale for SUS scores was developed where this average score would be a C [13]. This grading scale can be seen in figure 7.1.



SUS Score Range	Grade	Percentile Range
84.1 – 100	A+	96 – 100
80.8 – 84	A	90 – 95
78.9 – 80.7	A-	85 – 89
77.2 – 78.8	B+	80 – 84
74.1 – 77.1	B	70 – 79
72.6 – 74	B-	65 – 69
71.1 – 72.5	C+	60 – 64
65 – 71	C	41 – 59
63.7 – 64.9	C-	35 – 40
51.7 – 62.6	D	15 – 34
0 – 51.7	F	0 – 14

Figure 7.1: Curved grading scale for SUS scores

The average SUS score for our implementation was 81,25. This is much higher than average, and would result in an A on the grading scale.

We did not have a lot of participants for the usability study, but according to Nielsen [6], we only need as few as 5 participants, or an even lower number for smaller projects, to find almost as many usability problems as we would find with a much larger number of participants.

We would therefore say that the main limitation of this study is not the low number of participants, but rather the background of the participants. As we can see in table 7.1 on page 39, the participants mostly had prior exposure to OptiqueVQS and/or an IT-related background, which can explain the very high

SUS scores. The target group of OptiqueVQS is domain experts without much technical knowledge, so our participants do fall outside of this group.

When we were comparing solutions and deciding which one of the four to implement in section 5.4 on page 28, we mentioned that we did think solution 1 and solution 2 were pretty similar in terms of functionality and usability. We did not have a clear preference, but ended up choosing solution 1, mainly because it required less clicks than solution 2. After the alterations we chose to make and had to make during the implementation process, this might not be the case anymore.

We could also see one of the participants pausing a bit before noticing he had to change his focus from the left pane to the right pane for the subclass selection, which would not have been an issue with solution 2. Whether or not we made the right choice between these two solutions is therefore a bit unclear.

As mentioned in section 7.1.4 on page 41, we also did discover some usability issues that were not directly related to our implementation. All four participants expressed that they wished the object property would be in focus instead of the class name, so this is something that should be considered. For the pop-up window for subclass selection, we should also consider adding another button for closing the window, as the X button in the corner is associated with cancelling.

Chapter 8

Future work

In this chapter we will talk about some of the main aspects of our implementation that are in need of improvement. We will also briefly discuss some open questions related to subclass selection that came up during our work, that could be interesting to look further into.

8.1 Need further work

8.1.1 Efficiency

As discussed in section 4.5 on page 16, we are aware that the method we use for finding object properties and data properties that should be shown as options can potentially be slow, especially for larger ontologies. This is something that should be looked further into, to see if there are more efficient methods to calculate the same results.

8.1.2 Subclass selection

For subclass selection, there are two main issues that are in need of improvement.

Multiple levels of subclass selection

Currently, it is only possible to select the first level of subclasses in an ontology. Many ontologies have class hierarchies that span across multiple levels, so it is important to be able access and select more than just the first level of subclasses. We had a partially working solution for this, which is described in section 6.2.2 on page 32 and section 6.2.3 on page 32. Hopefully, this partially working solution is something that can be built upon and improved in the future.

Update property lists after subclass selection

Currently, the list of possible object properties is not updated after a subclass has been selected. The same goes for the list of possible data properties. This can result in properties that no longer should be available as an option still being possible to select.

Let's say we have selected the class Human. One of the object properties that are available to add for the class Human is hasDriversLicense. The domain of hasDriversLicense is Adult, which is a subclass of Human. Because we have not yet chosen a subclass, the Human could still be an Adult, so hasDriversLicense should be available as an option. If we now select the subclass Minor, which is disjoint with Adult, we should no longer be able to select the object property hasDriversLicense.

Currently, this does not happen. Options that should disappear after a subclass restriction is made are still available. For the initial classes that are added to the graph as new nodes, the available options are always correct. The problem is that the options are not updated whenever a subclass is selected, so finding a way to make this happen would be a big improvement.

8.2 Open questions

8.2.1 Suggesting more specific classes

What should happen if we have a general class and then select an object property with a specific domain, should we get a suggestion to change the general selected class to that specific class?

Let's say we have selected the class Human, and then we add the object property hasDriversLicense. The domain of hasDriversLicense is Adult, which is a subclass of Human. Because of this, we know that the selected class (Human) have to be an Adult. Should we then get a suggestion to change the class from Human to the more specific subclass Adult? Or should the change happen automatically?

And if we don't change the class, or suggest to change it, should we make the now invalid choices impossible to select? Let's continue with the same example; if we know, based on selections made, that the only possible subclass of Human that can be selected is Adult, should the other choices be hidden, or not selectable?

8.2.2 How to deal with changes in the query?

Another thing to think about is how to deal with changes that are made in the middle of the query that is being built. Some changes could possibly make part of the query invalid.

Let's continue with the same example as above, where we have selected the class Human, and added the object property hasDriversLicense. The domain of hasDriversLicense is Adult, which is a subclass of Human. Let's say we now go back to the Human node and select the subclass Minor. The subclass Minor is disjoint from Adult and should not be able to have the object property hasDriversLicense.

What should happen in these situations? Should we be unable to select the subclass Minor, or get a message telling us to delete the hasDriversLicense edge before we can change the subclass to Minor? Or should the system let us make the change, and then tell us that the edge hasDriversLicense is now invalid and must be removed, or even remove it for us automatically?

Chapter 9

Conclusion

In this thesis, we have worked on both the backend and the frontend of OptiqueVQS to create a solution that is more suited for handling subclasses.

For the backend, we have used disjointness reasoning to implement a new method of finding which properties should be shown as options for a given class.

For the frontend, we have explored different possibilities for presenting subclass selection. Based on analysis and findings from a user study, we implemented the solution we believed would result in the best usability.

After implementing changes to both the backend and frontend, we conducted a usability study with tasks focused on subclass selection. After completing the tasks, the participants were asked to fill out a System Usability Scale (SUS), where they rate statements about the usability. The average score ended up being very high, and would result in an A on a grading scale for SUS scores. While this result might be slightly skewed because the participants were not part of the main target group of OptiqueVQS, we still believe the result shows us that our solution has a high usability.

Bibliography

[1] Sean Bechhofer and Ian Horrocks. "Driving User Interfaces from FaCT". In: *Proceedings of the 2000 International Workshop on Description Logics (DL2000), Aachen, Germany, August 17-19, 2000*. Ed. by Franz Baader and Ulrike Sattler. Vol. 33. CEUR Workshop Proceedings. CEUR-WS.org, 2000, pp. 45–54. URL: `http://ceur-ws.org/Vol-33/Bechhofer45-54.ps`.

[2] John Brooke. "SUS: A quick and dirty usability scale". In: *Usability Eval. Ind.* 189 (Nov. 1995).

[3] Tom Fredrik Christoffersen. *SPARQL Extension Ranking - Collaborative filtering for OptiqueVQS-queries*. eng. Tech. rep. University of Oslo, 2020. URL: `https://www.duo.uio.no/handle/10852/78711` (visited on 06/10/2022).

[4] Nicola Guarino, Daniel Oberle, and Steffen Staab. "What Is an Ontology?" In: *Handbook on Ontologies*. May 2009, pp. 1–17. DOI: `10.1007/978-3-540-92673-3_0`.

[5] Steve Harris and Andy Seaborne. *SPARQL 1.1 Query Language*. Mar. 2013. URL: `https://www.w3.org/TR/2013/REC-sparql11-query-20130321/` (visited on 06/12/2022).

[6] Jakob Nielsen. *How Many Test Users in a Usability Study?* en. Mar. 2012. URL: `https://www.nngroup.com/articles/how-many-test-users/` (visited on 06/11/2022).

[7] Jakob Nielsen. *Thinking Aloud: The #1 Usability Tool*. en. Jan. 2012. URL: `https://www.nngroup.com/articles/thinking-aloud-the-1-usability-tool/` (visited on 06/09/2022).

[8] *OWL 2 Web Ontology Language Primer (Second Edition)*. URL: `https://www.w3.org/TR/2012/REC-owl2-primer-20121211/` (visited on 06/12/2022).

[9] *OWL API Wiki*. en. Oct. 2017. URL: `https://github.com/owlcs/owlapi` (visited on 06/12/2022).

[10] *OWLAPI Examples*. original-date: 2013-02-17T11:51:14Z. June 2022. URL: https://github.com/owlcs/owlapi/blob/1c5b14cfbc80e05735ec5738b0fee2b29656701a/contract/src/test/java/org/semanticweb/owlapi/examples/Examples.java (visited on 06/11/2022).

[11] *RDF 1.1 Concepts and Abstract Syntax*. Feb. 2014. URL: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (visited on 05/16/2022).

[12] *RDF 1.1 Primer*. June 2014. URL: https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/ (visited on 06/10/2022).

[13] Jeff Sauro and James R. Lewis. "Chapter 8 - Standardized Usability Questionnaires". In: *Quantifying the User Experience*. Ed. by Jeff Sauro and James R. Lewis. Boston: Morgan Kaufmann, 2012, pp. 185–240. ISBN: 978-0-12-384968-7. DOI: https://doi.org/10.1016/B978-0-12-384968-7.00008-4. URL: https://www.sciencedirect.com/science/article/pii/B9780123849687000084.

[14] Jeff Sauro and James R. Lewis. "Correlations among Prototypical Usability Metrics: Evidence for the Construct of Usability". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '09. event-place: Boston, MA, USA. New York, NY, USA: Association for Computing Machinery, 2009, pp. 1609–1618. ISBN: 978-1-60558-246-7. DOI: 10.1145/1518701.1518947. URL: https://doi.org/10.1145/1518701.1518947.

[15] Ahmet Soylu et al. "OptiqueVQS: A visual query system over ontologies for industry". en. In: *Semantic Web* 9.5 (Aug. 2018). Ed. by Freddy Lecue, pp. 627–660. ISSN: 22104968, 15700844. DOI: 10.3233/SW-180293. URL: https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/SW-180293 (visited on 05/19/2022).

[16] Rudi Studer, V. Richard Benjamins, and Dieter Fensel. "Knowledge engineering: Principles and methods". In: *Data & Knowledge Engineering* 25.1 (1998), pp. 161–197. ISSN: 0169-023X. DOI: https://doi.org/10.1016/S0169-023X(97)00056-6. URL: https://www.sciencedirect.com/science/article/pii/S0169023X97000566.

[17] The W3C SPARQL Working Group. *SPARQL 1.1 Overview*. Mar. 2013. URL: https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/ (visited on 05/16/2022).

[18] W3C OWL Working Group. *OWL 2 Web Ontology Language Document Overview (Second Edition)*. Nov. 2012. URL: https://www.w3.org/TR/2012/REC-owl2-overview-20121211/ (visited on 05/16/2022).