Collaborative Query Formulation in a Visual Query System

Master’s thesis

Yasar Mumtaz
Autumn 2015
Acknowledgements

First, I would like to express my sincere gratitude to my supervisors Martin Giese and Ahmet Soylu for their guidance, support and engagement throughout this thesis work.

Furthermore, I would like to thank my managers at Visma Consulting for being kind enough to provide me a flexible working environment without which it would have been very difficult to accomplish this task.

Last but not the least I would like to thank my family, my parents, my wife and my lovely kids for supporting me spiritually throughout writing this thesis and my life in general.
Abstract

This work has been carried out within Optique, a European research project that aims to provide non-IT end users an access to information in large and complex data stores. A prototypical query formulation interface OptiqueVQS has been implemented as a part of Optique project. OptiqueVQS is an ontology-based visual query system that provides end-users an opportunity to formulate queries according to their needs without having any technical expertise. This interface supports visual query formulation where users can navigate among different concepts and create a graph/tree structure. This interface then converts these visual queries into SPARQL queries and retrieves information from underlying data sources.

What is missing is a possibility to author/edit/formulate SPARQL queries in textual format. A system where queries can be formulated in both textual and visual format and are in comprehensive synchronization with each other will enable domain experts and technical specialists to formulate complex queries in collaboration. This thesis aims to develop such a system.
# Contents

Acknowledgements ........................................................................................................ iii

Abstract ......................................................................................................................... v

List of figures .................................................................................................................. xi

List of tables ..................................................................................................................... xiii

I Introduction ..................................................................................................................... 1

1 Background ..................................................................................................................... 3

1.1 Introduction .............................................................................................................. 3

1.2 Problem statement ................................................................................................... 4

1.3 Optique ..................................................................................................................... 6

1.4 Optique VQS .......................................................................................................... 9

2 Semantic technologies .................................................................................................. 13

2.1 Resource description framework ........................................................................... 13

2.2 Resource description framework – Schema (RDFS) ............................................... 15

2.3 What is an Ontology? ............................................................................................. 15

2.4 Web ontology language (OWL) ............................................................................. 16

2.5 SPARQL .................................................................................................................. 17

3 Collaborative Query Formulation ................................................................................. 21

4 Related Work ................................................................................................................. 23

II The Project .................................................................................................................... 25

5 Architecture and design ............................................................................................... 27

6 Tools and Libraries ....................................................................................................... 29

6.1 jQuery Mobile .......................................................................................................... 29

6.2 JavaScript InfoVis Toolkit ..................................................................................... 29

6.3 SPARQL.js ............................................................................................................. 30
SPARQL to VQS

7.1 Editable query area

7.2 Cleaning SPARQL

7.3 Parsing SPARQL

7.4 Pre-Processing SPARQL

7.4.1 Processing undefined types

7.4.2 Processing cycles

7.5 Validation of ontology

7.5.1 Core concepts

7.5.2 Concept fields

7.5.3 Adjacencies

7.6 Transformation of parsed JSON to Optique JSON

7.6.1 Start Node

7.6.2 Processing typed variable (Concepts)

7.6.3 Aggregates

Alternative approach for transformation

III Conclusion

Summary

Discussion and evaluation

Conclusion

11.1 Future Work

Bibliography

IV Appendices

Appendix A

Appendix B
14.1  Simple query with two concepts ................................................................. 71
14.2  Query with two concepts and one filter ...................................................... 77
14.3  Query with inverse relationship ................................................................. 85
14.4  Query with aggregates ............................................................................... 91
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>OptiqueVQS with textual and visual modes</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Simple case</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>Complex case</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Optique architecture</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>OptiqueVQS Visual Query</td>
<td>9</td>
</tr>
<tr>
<td>1.6</td>
<td>OptiqueVQS SPARQL Query</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>RDF graph</td>
<td>15</td>
</tr>
<tr>
<td>5.1</td>
<td>Architecture</td>
<td>28</td>
</tr>
<tr>
<td>7.1</td>
<td>Start node</td>
<td>41</td>
</tr>
</tbody>
</table>
List of tables

2.1: RDF statements ........................................................................................................ 14
2.2: Triple pattern ............................................................................................................ 18
2.3: Graph pattern ........................................................................................................... 18
2.4: SELECT query ............................................................................................................ 19
7.1: ContentEditable div .............................................................................................. 32
7.2: Clean SPARQL .......................................................................................................... 32
7.3: SPARQL query 1 ...................................................................................................... 33
7.4: JSON 1 .................................................................................................................... 34
7.5: Missing types .......................................................................................................... 35
7.6: Missing types added ............................................................................................... 36
7.7: Similar types ............................................................................................................ 36
7.8: Algorithm for processing cycles .............................................................................. 37
7.9: Type added to handle cycles .................................................................................. 38
7.10: Similar types ......................................................................................................... 38
7.11: Same nodes array ................................................................................................. 39
7.12: Same types filter ................................................................................................... 39
7.13: First node JSON .................................................................................................... 42
7.14: Metadata for concept ............................................................................................ 43
7.15: Type triple ............................................................................................................... 43
7.16: SPARQL outputs .................................................................................................... 44
7.17: JSON outputs ......................................................................................................... 45
7.18: SPARQL Filter ....................................................................................................... 46
7.19: SPARQL Filter in JSON ......................................................................................... 46
7.20: String comparison filter ....................................................................................... 47
7.21: SPARQL adjacencies .............................................................................................. 48
7.22: SPARQL adjacencies RJSON ................................................................................. 48
7.23: SPARQL adjacencies Optique JSON ..................................................................... 48
7.24: SPARQL with inverse properties ......................................................................... 49
7.25: RJSON with inverse properties ............................................................................ 49
7.26: Optique JSON with Inverse properties ................................................................. 50
7.27: SPARQL with aggregate ................................................................. 51
7.28: Aggregates in RJSON ................................................................. 51
7.29: Start node with aggregate information ........................................ 52
8.1: Text to tree JSON structure ......................................................... 53
8.2: Text to tree calling tree methods in OptiqueVQS ......................... 54
Part I

I Introduction
Chapter 1

Background

1.1 Introduction

A number of organizations need to retrieve information from their databases on regular basis for different purposes such as reporting, decision making, planning future projects etc. Employees of these organizations possesses a very good knowledge of their respective domains but when it comes to extracting information from huge data stores, they face some limitations. As most of them are only domain experts, they are restricted to use pre-defined queries that are setup for them by technical experts. When there is a need to extract some information that cannot be extracted by predefined queries they have to involve technical experts. This approach create different problems such as delay in process of retrieving information [1]. This delay can be because of factors like availability and access to technical experts who may or may not be employees of the same company. Significant extra cost is another problem.

Computer scientists have long been researching on this issue and visual query formulation is one of the areas explored and discussed in this particular context [2]. By visual query formulation, researchers are trying to create a solution where domain experts can extract information from databases according to their requirements without any technical knowledge. The advantage of using a visual query system is that it follows the direct manipulation idea where domain and query language are represented with a set of visual elements [1] [3]. Domain experts can easily identify concepts and relationships that are represented by these visual elements and they can navigate among them to formulate a query. The visual queries created in these systems are converted into actual queries that are executed against data sources for retrieving information.

A lot of research has been done in the area of visual query formulation for relational data. Most of these researches were able to implement the basics of visual query formulation but they were not very successful mainly due to the abstraction levels such as database schemas or object-oriented models. In recent researches, it has been found that using ontologies for visual query formulation has great potential as ontologies are close to reality and they can work as a layer between relational data and visual query interface. This approach of accessing data via ontology is known as ontology-based data access and is often referred by acronym OBDA [1] [4] [5].
OptiqueVQS is one such system that provides domain experts a possibility to construct visual queries for retrieving information according to their needs. It also provides a feature to view SPARQL query generated from VQS although it is not possible to make changes in textual format in its current implementation [1].

### 1.2 Problem statement

A visual query system such as one mentioned above can be much more effective if it is possible for technical experts to make changes in textual version of these queries in addition to domain expert’s manipulation of visual version.

![Figure 1.1: OptiqueVQS with textual and visual modes](image)

In a situation where domain experts have created a query according to their requirement and are not getting their desired results, technical experts can have a quick look into textual query and fix it for them. In this way, domain experts and technical experts formulate a query in collaboration to achieve the desired results. There are many advantages of having a solution like this for
example, textual queries are very easy to customize and exchange when compared to visual queries. If a domain expert is struggling to get the desired results, he/she can easily send the textual query to a domain expert who can fix the query and send it back to domain expert who can then load it in visual format. Currently OptiqueVQS only provides a feature to generate SPARQL query from Visual query (VQS to SPARQL) as shown in Figure 1.1.

This thesis aims to develop a solution where it will be possible for technical experts to create new SPARQL queries and to update the existing ones. On switching to visual query mode, latest version of SPARQL query will be transformed into visual query (SPARQL to VQS). A synchronization between textual and visual modes will be maintained throughout query formulation process. This synchronization will make it possible to switch between these two modes at any point.
1.3 Optique

Data is exponentially growing and almost every big organization needs to access its enormous amount of data for analysis and value creation. The amount and complexity of data is growing very fast which is making it difficult to access important data in reasonable time. Optique\(^1\) is a research project that promises to bring revolutionary changes to access big data [6]. With existing technologies domain experts (Engineers) have to rely on pre-defined queries to access the information as shown in figure 1.2 [7]. If there are multiple data sources then it is also mandatory for domain experts to have a knowledge of them so that they can submit query against the appropriate one.

**Simple case:**

![Simple case diagram]

Figure 1.2: Simple case

In addition to that whenever there is a need to access data that is not available via predefined queries, technical experts have to be involved which leads to delay in accessing the data and hence overall effectiveness is compromised as shown in figure 1.3 [7].

**Complex case:**

![Complex case diagram]

Figure 1.3: Complex case

---

\(^1\) http://optique-project.eu/
These are the bottlenecks, which limits the exploitation of enormous amount of data or “Big data”. Optique will accomplish its goals of removing these bottlenecks by providing end-to-end semantic connections between data sources and users, enabling users to formulate spontaneous queries using familiar vocabularies, integrating underlying data sources in seamless way and exploiting parallelism for quick response time among others [7].

From above discussion we have reached at the conclusion that major bottleneck in end-user access to corporate data is translation of end-user information requirements into queries that can be executed against data sources. Optique uses Ontology-Based Data Access (OBDA) approach that has the potential to remove this bottleneck by automating the query translation process [8] [9]. An ontology in such an approach is a formalization of end-user vocabulary to address problem domain. It is independent of underlying data storage. End-users formulate queries
according to terms and concepts that are familiar to them. IT experts produce a set of mappings that describe relationships between ontology terms and their respective representation in data sources. With user-formulated query, ontology and a set of mappings an algorithm transforms user query into a query that can be executed against the appropriate underlying data source. The results are then returned to end user [10].

An architecture diagram of Optique is shown in Figure 1.4 above [11].
1.4 Optique VQS

A user interface mashup is a collection of simple rules that help creating responsive applications using client side technologies. The major advantage of using UI Mashups is that they provide “a unified interaction experience over a common graphical space for a collection of distributed digital entities and necessary affordances to blend functionalities of these entities and to stimulate exploration of new possibilities” [12].

OptiqueVQS is a graphical user interface based on same architecture. Widgets in these kind of systems are interconnected with each other via client communication channel. When a user perform an action on any of these widgets it deliver events to other related widgets and graphical user interface is updated accordingly [1].

![Figure 1.5: OptiqueVQS Visual Query](image)

There are currently three main widgets in OptiqueVQS. The first widget W1, on the bottom left of the Figure 1.5 is a menu-based representation of query by navigation interaction paradigm (QbN). End-users can navigate through different concepts according to their relationship hierarchy. The widget on the bottom right W2 shows the attributes and constraints related to selected concept from widget W1. In this widget, users can select attributes to be included in result set and can apply constraints. The widget on the top W3 shows the visual query formed so
far with help of graphical symbols. The query is shown in a directed tree format from left to right where concepts are linked with each other by one direction links.

When OptiqueVQS starts for the first time, it asks users to select an ontology, against which queries are to be formulated. It then redirects to main interface where canvas in W3 shows a default concept, which is a dummy concept with text “Untitled Query”. At this point W1 is loaded with all the core concepts in the underlying ontology. User selects a core concept and a graphical shape (node) is created on W3, this node got the focus and is shown in orange color. At this point, W2 shows all the attributes related to selected concept. User can select these attributes if he want them to be included in result set. In the same window user can also apply constraints by using form elements. These attributes and constraints are then displayed on selected concept (node) in W3 with prefixes “o” for output attributes and “c” for constraints. W1 at this point is loaded with all the related concepts to the one currently selected. This help to reduce the number of navigational levels. Now if user will select another concept from the W1, this will be created in W3 next to the one created in previous step with a connection. Same steps can be followed to include further concepts. The connections between nodes are one directional, which also means that query will always be read from left to right. An example query with Wellbore and Field concepts has been shown in Figure 1.5 above.

It is also possible for end users to delete nodes in W3 by first clicking on the “Delete node” button and then clicking on the concept that is to be deleted. “Undo” and “Redo” features provides users a flexibility to get back to an earlier state. A query can be saved by clicking on the “Save Query” button while clicking on “Stored queries” shows a list of saved queries that can be loaded if desired.

The user can also switch to SPARQL mode by clicking on “SPARQL Query” button in W3. In this mode, a textual SPARQL query is displayed to user as shown in Figure 1.6. Currently this textual query cannot be edited and is only for learning and testing purposes for advanced users [1].
Figure 1.6: OptiqueVQS SPARQL Query

```sparql
PREFIX ms1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ms2: <http://ows.it.aoe.sns.bg/odp-v20p>

SELECT ?s1 ?d2 WHERE {
  ?s1 ms1:,type ms2:Wellbore.
  ?s2 ms1:type ms2:Field.
  ?s1 ms2:developmentWellboreForField ?d1.
  ?s1 ms2: name ?s1.
  ?s2 ms2: wellboreWellType ?d2.
}
```
Chapter 2

Semantic technologies

The word semantics\(^1\) comes from Greek word *semanticos*, which means “significant meaning” and is used to describe study of meaning. This term originally belongs to philosophy but also frequently used in computer science. A software technology that exploits the meanings of the information by involving the use of a certain knowledge model is defined as semantic technology. Inference engines, ontologies, agents and rule engines are examples of semantic technologies. Semantic technologies are capable for many interesting things such as integration of information, information fusion, semantic search, information presentation and navigation. It is still a developing field and there are many issues as well, some of these are ontology mapping problem and issues related to reasoning [13].

Semantic web is the strongest area of semantic technologies today and a lot of research is going on in this area. According to Berner-Lee, semantic web is an extension of current World Wide Web, where information gives well defined meanings enabling computers and people to cooperate with each other [14].

The main purpose of the semantic web is to add meanings to current Web, that are understandable to computers, which will help completing tasks that are otherwise accomplished manually. As a whole, we can say that it is a collection of technologies and standards making it possible for computers to understand meanings of information on the web [15].

Following is a brief introduction of major semantic technologies and standards that are related to this thesis work.

2.1 Resource description framework

Resource description framework [16] was created by W3C in 1999 as a metadata encoding standard. It is known as building block of semantic web because knowledge represented by RDF follows a standard structure, which allows easy exchange of machine understandable information on web. It represents scattered information and knowledge in such a way that computers can process it in an extensible manner. It has the same role for Semantic Web as HTML is for Web [15]. RDF uses an abstract model to represent knowledge. It decomposes the information into

\(^1\) https://en.wikipedia.org/wiki/Semantics
small chunks with the help of simple rules about the meanings of these chunks. The objective is
to provide simple and flexible way to express facts but at the same time they should be so well
structured that computers can process the expressed knowledge [15]. All the information in RDF
is articulated with the help of a triple pattern also known as a statement. A triple or a statement
consists of a subject, a predicate and an object in the form of Subject-Predicate-Object and this
form never changes. The subject and object are names of two things and predicate is the
relationship between them. Here are some example statements.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>USA</td>
<td>has capital</td>
<td>Washington D.C.</td>
</tr>
<tr>
<td>USA</td>
<td>has president</td>
<td>Barak Obama</td>
</tr>
<tr>
<td>Barak Obama</td>
<td>born year</td>
<td>1961</td>
</tr>
</tbody>
</table>

Table 2.1: RDF statements

A subject or an object in a statement can be concrete or abstract but in both cases, it is a resource.
To avoid ambiguity and name clashes the name of a resource must be global and should be
identified by a Uniform Resource Identifier [10]. The name of predicate must also be global and
is identified by a URI.

Example URI’s are
- USA: http://dbpedia.org/resource/USA
- has capital: http://dbpedia.org/ontology/capital

An RDF model can be expressed as either a collection of statements or a graph. A graph
representation of Table 2.1 is shown in Figure 2.1 below. The linking structure forms a directed
labelled graph with nodes as resources and edges as relationships. The elements of an RDF triple
are URI references, blank nodes and literals. URI references are discussed above, literals
contains values such as string or integers while a blank node is nameless node without URI. RDF
has different representations such as XML serialization, N3, turtle and N-Triples.
2.2 Resource description framework – Schema (RDFS)

RDFS is a language used to create a vocabulary, which is most of times domain-specific. This vocabulary is then used to create RDF documents, so that everything we describe in our document has a reason to be described [15].

According to W3C “RDFS is recommendation from W3C and it is an extensible knowledge representation language that one can use to create a vocabulary for describing classes, subclasses and properties of RDF resources” [17]. RDFS terms are defined by URI’s just like RDF and they can be divided into classes and properties. A group of resources is defined as a class. A class has a set of instances also called as members of the class. According to W3C recommendations rdfs:Resrouce, rdfs:Literal, rdfs:Datatype and rdfs:Class are classes [17]. RDF property defines a relationship between a subject resource and an object resource. Properties available in RDFS are rdfs:range, rdfs:domain, rdfs:subPropertyOf, rdfs:subClassOf, rdfs:lable and rdfs:comment [17].

2.3 What is an Ontology?

Ontology based data access is focus of Optique project and this thesis is a part of it, therefore, it is necessary to briefly describe about the concept of ontologies and how they are useful.

A common set of terms used to describe a domain is defined as an Ontology [18]. An ontology is specific to one particular domain and characterizes an area of knowledge. A domain means a specific subject for example education, sports, medicine etc. An ontology contains concepts and relationships among these concepts. Concepts are also known as classes or terms. The relationships between these concepts can be stated with the help of hierarchical structures, where
super-classes defines more general concepts, whereas sub-classes defines more specific and detailed concepts. These specific concepts inherits attributes and properties from their super concepts. Apart from hierarchical relationships, there can be property relationships among concepts as well. Once concepts and relationships are clearly defined, the knowledge represented by the ontology becomes easier for machines to understand and process [15].

Ontologies can be compared to other models such as databases, object models, business rules and XML schemas. They can be compared with vocabularies and taxonomies as well. They have things in common with all these and have features that are different [13].

There are several benefits of using ontologies such as

- They can be linked with each other, making it possible to develop them in components.
- A metadata of high level precision can be defined.
- Reuse of knowledge across different models
- Querying to analyze domain knowledge.

2.4 Web ontology language (OWL)

There are several languages used to create ontologies and OWL is the most popular among them. OWL stands for web ontology language and is a language built on description logics. It becomes a W3C recommendation in 2004. Just like the RDF schema, its main purpose is to define ontologies. The difference with RDFS is its capabilities to express much more complex and detailed relationships. We can conclude that OWL consists of RDFS and new constructs to deal with complex situations. The history of OWL development is dated back to 1990s when many researches were conducted to find out how knowledge representation from artificial intelligence can be applied to web [15].

OWL 1 and OWL 2 are two major versions of web ontology language. OWL 2 is the latest version, which includes everything that was in OWL 1 making it a subset of OWL 2. OWL 1 is still very important since a number of well-known ontologies were written in this language and a vast majority of ontology engineering tools only supports creating ontologies using this. OWL 2 offers some exciting features such as syntactic sugaring to make statements easier to construct, improve expressiveness, extended support for datatypes, annotations and sublanguages [15].

OWL 2 has two alternative methods to assign semantics to ontologies namely OWL 2 DL and OWL 2 Full. OWL 2 DL talk about ontologies that are interpreted by using direct semantics
while OWL 2 Full talk about ontologies that are interpreted by using RDF-based semantics. Description logic is used for assigning meanings to ontologies in OWL 2 DL and it is considered as a syntactically limited version or a sublanguage of OWL 2 full. OWL 2 DL makes implementation of reasoners simple and easy [15].

In addition to the two alternative approaches for assigning semantics, OWL 2 also specifies different language profiles such as QL, EL and RL. A profile is also called as a sublanguage or a fragment in computational logics, in this scenario it is trimmed down version of OWL 2 that “trades some expressive power for the efficiency of reasoning” [19].

EL profile is useful for applications working with very complex ontologies having large number of classes and properties. It has polynomial time reasoning. RL profile is designed for applications requiring scalable reasoning without exploiting too much expressive power. It is compatible with rule base engines [19].

QL is specifically designed for applications requiring efficient database integration. Query answering against large data sources is the most important reasoning task in these applications. This profile guarantee polynomial-time performance. The language constructs supported by this profile represent key features of entity relationships and UML, which make it possible to use this profile as a high-level database schema language. QL is complaint with OBDA approach that has been used in Optique [15] [19].

2.5 SPARQL

SPARQL is a query language and a data access protocol for RDF. It stands for SPARQL Protocol and RDF Query Language. It can be used to query diverse data sources both native RDF and those viewed as RDF with the help of a mapping layer. It has the power to query optional and required graph patterns together with conjunctions and disjunctions [20].

From brief discussion about RDF, OWL and ontologies above we have an understanding that we can create data models and formats that can be processed by computer applications. To search and analyze data from these RDF documents or ontologies we need a query language in which we can submit a query to get the desired results. SPARQL is one such query language designed to serve this purpose [15]. A SPARQL query is executed against an RDF dataset through an interface known as a SPARQL endpoint, which can be either standalone or a web application. Following are four different forms of a SPARQL query:
Currently we are only dealing with SELECT queries in OptiqueVQS and therefore structure and working of only SELECT queries will be discussed here. All the query forms in SPARQL are based on two core concepts, the triple pattern and the graph pattern.

As we have discussed in RDF section a triple pattern consists of a subject, a predicate and an object in the form of Subject-Predicate-Object. The difference between a RDF triple and an SPARQL triple is that in a SPARQL triple all or any of the subject, predicate and object can be a variable. Here is how a triple pattern can look like in a SPARQL query [15].

```
?c1 ns1:type ns2:Wellbore
```

Listing 2.2: Triple pattern

A variable in SPARQL is prefixed with either a ? or a $. I will be using ? as variable prefix in my examples. In above example ?c1 is a variable which has type Wellbore.

A graph pattern is another very important concept in SPARQL. It is used for selecting triples from a RDF graph and can specify very complex selection rules as compared to triple pattern. A collection of triple patterns makes a graph pattern, which is specified with the help of curly brackets {} [15].

```
{ 
   ?c1 ns1:type ns2:Wellbore.
   ?c1 ns2:wellboreSeismicLocation ?a1.
}
```

Listing 2.3: Graph pattern

Now when we have an understanding of triple and graph patterns in SPARQL we will discuss SELECT queries, which are fundamental part of this thesis work. A SELECT query has following structure.

BASE (base directive)

PREFIXES (List of prefixes)
SELECT (List of variables)
FROM (graph for search)
WHERE (query pattern)
ORDER BY (query modifiers)

The BASE directive combine the relative IRIs. There can be multiple prefixes in a SELECT query and these prefixes are used for URI abbreviations. The SELECT part contains a list of variable bindings and it specifies what to pick from a result set. The FROM clause is optional in some cases and it specifies the endpoint against which the graph search is to be performed. The WHERE is a mandatory clause that tells the endpoint what to query from data graph. WHERE keyword is optional. The clauses that comes last such as ORDER BY and LIMIT are used for organizing and restricting the results. They are called query modifiers [15].

Here is an example SELECT query that will return 100 or less results with all the Wellbores with their seismic locations and statuses.

```sparql
PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
SELECT ?c1 ?a1 ?a2 WHERE {
    ?c1 ns1:type ns2:Wellbore.
    ?c1 ns2:wellboreSeismicLocation ?a1.
} LIMIT 100
```

Listing 2.4: SELECT query
Chapter 3

Collaborative Query Formulation

This thesis work is about implementing a collaborative query formulation solution in a VQS. Therefore, it is necessary to have a basic understanding of what is collaborative query formulation and how it works.

Collaborative query formulation is a process where a group of people work together to construct a query to accomplish a task. Active and passive collaboration are two main types of collaborative query formulation.

“Passive collaboration often refer to a collection of techniques that help users formulating queries to meet their information needs by harnessing other user’s expert knowledge and search experience” [21]. Queries are expression of information needs and they provides a lot of information that can be used to guide other users formulating similar queries. This can be achieved by storing large amount of queries that can then be mined using different data mining algorithms to find out similar patterns and relationships. This will not only help users formulating queries according to their information needs but can also suggest related information [21]. Online question-answering communities is an example of passive collaboration. Questions are posted by users looking for a solution to a specific problem and they are visible to all those visiting the site. Since most of the problems are not encountered for the first time so there are some people, who can suggest a solution or maybe an answer is already there. In this way, individual users not only find answers to their questions but can also refine and reformulate their problem statements. Stack overflow is one such community where programming related problems are discussed while Yahoo Answers is for general daily life problems [22].

In addition to passive collaboration where interaction logs from many users are mined to fetch information that can help creating new queries, research is also going on for providing more active collaborative experiences among users in a group. In this case, multiple users become involved to accomplish a single task. One drawback with this kind of collaboration is that it becomes very specialized since people involved are supposed to possess knowledge about a special domain.

Another kind of active collaboration can be among more than one methods of formulating a query when they are provided in same solution. Visual query systems are a popular approach
where non-technical domain experts formulate queries according to their needs without involving technical experts. Most of the time users are only provided with a graphical interface where they navigate among different concepts, select attributes, apply constraints and then finalize query. This query is then converted into an actual database query behind the scenes to fetch the data. Sometimes it is also possible for end-users to see their formulated query in textual format. It can be very useful for advanced users to make changes in both textual and visual format and formulate a query in collaboration of visual and textual format.
Chapter 4

Related Work

There are a number of code generating tools available for almost all the major programming languages. These tools are used in model driven architecture where user/programmer creates his model in some modeling language such as UML and these tools generate respective code that can be further extended to solve programming tasks. One example of the Java code generator is Acceleo\(^1\), which is available as a plugin in Eclipse environment. It is an implementation of MOFM2T standard from OMG. Microsoft Visual studio 2013\(^2\) also provides a feature to generate C# code from UML class diagrams.

While there are dozens of code generators available, there are only a few where it is possible to recreate models from the generated code. Visual-Paradigm\(^3\) is a design and management tool for Business IT development and it offers features of code engineering as well. Users creates a UML class diagram and generate code from it he/she can then make changes in the code and these changes will be synchronized with UML diagram. Borland\(^4\) Together is also another visual modeling software that generates code from models and keeps a synchronization between model and code.

There are also a number of open source and proprietary tools available for visual query formulation of SQL queries. SQLeo\(^5\) Visual Query Builder is a SQL tool that enables users to create visual queries. This tool also allow reverse engineering of database model. Other examples of query builders are SQuirreL SQL, Active Query Builder, Flyspeed, Oracle SQL developer, Toad, DbVisualizer etc.

There are also some visual query builders for building a SPARQL query. Interactive SPARQL (iSPARQL) Query Builder\(^6\) is one of them, it enable users to build a SPARQL query in browser. Another visual query builder for Drupal has also been presented where users or programmers of Drupal who are not familiar with SPARQL can formulate a query by drag and drop [23].

---

1. [https://eclipse.org/acceleo/](https://eclipse.org/acceleo/)
In visual query domain, there are two major categories for querying semantic data sources. One of them is visual query language (VQL) and other is visual query system (VQS). As the name suggest VQL is a language and therefore has a formal syntax and notation whereas VQS uses a system of interactions. OptiqueVQS follows VQS approach. The noticeable examples of VQSs are gFacet [24], SparqlFilter-Flow [25], Konduit VQB [26], and Rhizomer [27]. Although some of the related work discussed above partially relates to our goal but none of them offers a complete solution for our needs.
Part II

II The Project
Chapter 5

Architecture and design

This thesis work is about adding extra features to OptiqueVQS that is an ontology-based visual query interface for big data [1]. There are no major architectural changes planned at project level for this work. The working of OptiqueVQS has been explained in detail in section 1.4 and a brief description of the architecture is given below.

OptiqueVQS follows an approach that is built upon widget-based user-interface mashups. The widgets in the mashup communicates with each other and behave like a single application while each one of these is a full-fledged client side application with less complexity and functionality [12]. A run-time environment manages these widgets and is responsible for communication among them. Every widget exposes its functionality to the run-time environment and whenever an action is performed on a widget it notify other widgets or the environment. On receiving an event/message from other widgets, a widget perform an appropriate action according to application logic. By using this approach, we divide a bigger and complex problem into smaller and less complex ones. This approach has many benefits such as simplicity, maintainability, reuse and different combinations of smaller components among others [1].

The main goal for this thesis is to develop a solution for formulating a query in both visual and textual format. The conversion from visual to textual format is already present in the system that make it is possible to generate SPARQL queries from visual queries while generating visual queries from SPARQL queries was not yet possible. A SPARQL query generated from OptiqueVQS is just a simple string of characters so it needs to be parsed to some convenient format before it can be processed to get back the visual query. As most of the logic in OptiqueVQS is on client side so a parser that can parse SPARQL queries to JSON on client side is an appropriate choice. Therefore, I have used a SPARQL parser by Ruben Verbog that is developed in Java Script [28]. For a given SPARQL query, this parser returns a JSON object that follows a standard structure and contains all the information present in query. OptiqueVQS also uses JSON behind the scenes for controlling visual representations. The main goal of this thesis work is to explore the possibilities for transforming JSON object achieved from SPARQL parser into Optique JSON. Figure 5.1 illustrates the architecture of this work.
Figure 5.1: Architecture

Switching from textual mode to visual mode includes following steps.

1. Convert textual SPARQL to JSON (RJSON)
2. Convert RJSON to OptiqueVQS JSON
3. Load visual query from OptiqueVQS JSON

RJSON is a name given to JSON object that I get from our SPARQL parser. “R” in RJSON refers to author of SPARQL parser Ruben Verborgh\(^1\). This name “RJSON” is occasionally used in implementation section when referring to parsed JSON object.

---

\(^1\)[http://ruben.verborgh.org/](http://ruben.verborgh.org/)
Chapter 6

Tools and Libraries

The implementation part in this work was mainly about front-end development in JavaScript, HTML 5 and CSS3. Since most of the backend development in main project is in java, Eclipse\(^\text{1}\) environment has been used with SVN\(^\text{2}\) as source control. I have written my JavaScript code, HTML and CSS in both notepad++\(^\text{3}\) and atom.io\(^\text{4}\) editors. Testing of implemented functionality was done in Google Chrome\(^\text{5}\) and Firefox\(^\text{6}\).

Following is a brief description of the libraries used for implementation of this thesis work.

### 6.1 jQuery Mobile

It is fast becoming a requirement to make the web applications available on all the platforms such as mobiles, tablets and desktop. To develop an application for each of these different platforms can be quite hectic, instead an application that is flexible enough to adapt according to platforms can be very useful. Some frameworks help us developing such applications and jQuery Mobile\(^\text{7}\) is one of best among them. It has been used in development of OptiqueVQS mainly due to diverse user groups of the system, who may prefer to use different platforms and they should get the same user experience.

### 6.2 JavaScript InfoVis Toolkit

Visualizations in OptiqueVQS are developed using JavaScript Infovis Toolkit\(^\text{8}\). This library helps to create interactive data visualizations. The top widget (W1) in OptiqueVQS is a canvas where a tree structure is created as user selects different concepts and properties from query by navigation and facet widgets. JIT library works with a JSON object behind the scenes, which makes it possible to recreate and save the queries.

---

\(^\text{1}\) [https://eclipse.org/juno/](https://eclipse.org/juno/)
\(^\text{3}\) [https://notepad-plus-plus.org/](https://notepad-plus-plus.org/)
\(^\text{4}\) [https://atom.io/](https://atom.io/)
\(^\text{5}\) [http://www.google.com/chrome/](http://www.google.com/chrome/)
\(^\text{7}\) [https://jquerymobile.com/](https://jquerymobile.com/)
6.3 SPARQL.js

This main objective of this thesis work was to find out a solution for generating visual queries from SPARQL queries. The SPARQL has a grammar in extended Backus-Naur Form\(^1\) that means we can define rules to parse a SPARQL query. For this work, I have used a SPARQL parser\(^2\) developed by Ruben Verbog. This parser is developed in JavaScript and perfectly suits my case as I am parsing SPARQL queries on client side. The parser also fully supports SPARQL 1.1 and is continuously reviewed and updated by the author.

\(^1\) [https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form](https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form)

\(^2\) [https://www.npmjs.com/package/sparql-parser](https://www.npmjs.com/package/sparql-parser)
Chapter 7

SPARQL to VQS

This chapter will discuss all the minor details for transformation of SPARQL queries into visual queries in OptiqueVQS. Before we start here is a brief summary about how visual queries are related to SPARQL queries in OptiqueVQS.

As of now, only linear and tree-shaped conjunctive queries are supported in OptiqueVQS. The query interface is based on an OBDA framework in the background that supports a conjunctive part of SPARQL 1.1 [29] and OWL 2 QL profile [19]. OWL 2 QL makes query answering possible by rewriting SPARQL queries into standard relational database queries [30]. OptiqueVQS uses graph-based organization of ontological elements for representing query and domain structures while ontological axioms are used for constraining the behavior of interface for extending available knowledge [31].

The basic structure of a SELECT query in SPARQL has been discussed in detail in section 2.5. All the queries generated by OptiqueVQS have a graph pattern that is a set of triple patterns. When a query is in editable state, it is composed of a partial graph pattern and a cursor position. In case a query is empty, cursor position is blank while it points to a concept variable in the query if it is not empty as shown in Figure 1.5: OptiqueVQS Visual Query. A new variable is created whenever a concept is selected from query by navigation widget (W1). Creation of new variable means a triple is added with type referring to selected concept. If cursor is pointing to a concept, clicking on an object property in facet widget (W2) adds an output triple for current concept. A selection or a projection operation on a data property in facet widget (W2) adds a new constraint triple. Finally, selection of a new concept from W1 while a concept is selected adds two triples to graph, one for defining the type of new concept and the other for specifying relationship between currently selected concept and newly created one [31].

7.1 Editable query area

OptiqueVQS is browser based solution where users performs all their actions inside a browser window. I have implemented a new feature that allows users to make changes in SPARQL queries. For making such changes I need an HTML element that supports content editing. There are two main choices in this regard. The first one is to use a text area element. The problem with
text area in my context is that it supports editing plain text only while the SPARQL query generated in OptiqueVQS has a certain formatting with indentations and hyperlinks for query concepts. I needs an element that supports manipulation of html and a div element with contenteditable attribute was perfect choice for me, as it not only supports formatting but html elements as well. HTML contenteditable attribute\(^\text{1}\) is introduced in HTML 5 and it specifies whether content of an element can be edited or not. The following markup is defined in my implementation to support editable area.

\[
<\text{div id="queryarea" contentEditable="true"}></\text{div}>
\]

Listing 7.1: ContentEditable div

### 7.2 Cleaning SPARQL

Once the user has decided to switch to visual query mode from textual mode the SPARQL query is fetched from markup for further processing. As mentioned above query area in OptiqueVQS supports html tags but a clean SPARQL query is needed for further processing. Therefore, I have to clean the query by removing html tags. This was done by using following built-in function in jQuery.

\[
\text{var cleanSparql} = $\text{("#queryarea").text();}
\]

Listing 7.2: Clean SPARQL

### 7.3 Parsing SPARQL

One of the most important part of this thesis work is parsing SPARQL query. The query I get from user is in html format. I cleaned it to remove html tags and get the pure SPARQL query. For further processing this query is needed to be parsed. As mentioned in section about libraries I have used a parser with the name SPARQL.js for this purpose. Here is an example SPARQL query.

```
PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
SELECT ?c1 ?a1 ?a2 WHERE {
```

\(^\text{1}\) [http://www.w3schools.com/tags/att_global_contenteditable.asp](http://www.w3schools.com/tags/att_global_contenteditable.asp)
For parsing above SPARQL query, I call parse method in SPARQL.js parser. The parser make sure that the query is syntactically correct and if not it returns a list of error messages with line numbers in the query. An icon becomes visible in case there are any errors and by clicking on the icon user can see the detailed error messages. Once the query is syntactically correct, the parser returns a JSON object generated from the query. SPARQL query from Listing 7.3 has returned following JSON object.

```
{
   "type": "query",
   "queryType": "SELECT",
   "variables": [
      "?c1",
      "?a1",
      "?a2"
   ],
   "where": [
      {
         "type": "bgp",
         "triples": [
            {
               "subject": "?c1",
               "predicate": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
               "object": "http://sws.ifi.uio.no/vocab/npd-v2#Wellbore"
            }
         ]
      }
   ]
}
```
Listing 7.4: JSON 1

The JSON object follows a certain structure according to different SPARQL constructs, which make it easier to understand and process further. The detail of each property listed above will come later in this section. In general, the JSON object tells us about the type of query (which will be SELECT in our case), a list of variables, a list of triples in where clause, a list of prefixes and other details according to complexity of query.
7.4 Pre-Processing SPARQL

Before starting to build visual query some pre-processing tasks must be performed on JSON object that has been gotten by parsing SPARQL query. These pre-processing tasks are discussed in detail in this section.

7.4.1 Processing undefined types

In OptiqueVQS, a node represents a type or a concept. Attributes, constraints and relationships for the concept are displayed on this node in visual mode. It means every type or concept must be defined in SPARQL query. There are times when not all types are explicitly defined but only relationship are defined between types and it is assumed that query processing will automatically create necessary types. I have implemented a feature for adding necessary types in pre-processing phase. Here is an example where there is no type defining triple for variable c2.

```xml
PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
PREFIX ns3: <http://www.optique-project.eu/well-ontology/>

SELECT ?c1 ?a1 ?c2 ?a2 WHERE {
  ?c1 ns1:type ns2:Company.
  ?c1 ns3:hasDimension ?c2.
}
```

Listing 7.5: Missing types

Correct type must be defined for c2 because the visual query is build type by type and all types must be present in JSON object. It can be seen that c1 is related to c2 by relationship ns3:hasDimension. To solve this problem I get all the types that are related with c1 from ontology. I then iterate over them to find out which one is connected to c1 by property ns3:hasDimension. This type is added to JSON object so that it models the SPARQL in listing 7.6. It can be seen that a new type triple is added to query defining Dimension type.

```xml
PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
PREFIX ns3: <http://www.optique-project.eu/well-ontology/>

```
SELECT ?c1 ?a1 ?c2 ?a2 WHERE {
   ?c1 ns1:type ns2:Company.
   ?c2 ns1:type ns3:Dimension.
   ?c1 ns3:hasDimension ?c2.
}
```

Listing 7.6: Missing types added

### 7.4.2 Processing cycles

Visual queries in OptiqueVQS are constructed as a directed tree from left to right. In SPARQL, one variable can be connected to another variable by two different relationships. Given a query graph pattern there exists a cycle, if there is at least one (undirected) path in the graph whose first node corresponds to the last.

However, there is no way to represent more than one relationships between two concepts in OptiqueVQS. Let suppose we have following scenario.

```
PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>

SELECT DISTINCT ?c1 ?c2 ?c3 WHERE {
   ?c1 ns1:type ns2:ShallowWellbore.
   ?c2 ns1:type ns2:Field.
   ?c1 ns2:shallowWellboreForField ?c2.
   ?c1 ns2:wellboreForField ?c2.
}
```

Listing 7.7: Similar types

We can see that c1 is related to c2 by both shallowWellboreForField and wellboreForField relationships. In such a case, an algorithm has been used to first detect cycles and then update the
JSON accordingly by creating necessary types and updating the relationships. Pseudo code for my algorithm is shown in listing below.

1. Get prefixes from the JSON object
2. Get type triples from the JSON object
3. Get output triples from JSON object
4. Initialize an array of relationship triples to be updated
5. Loop over type triples
   6. Get triple subject of first triple
   7. Get relationships triples for current type triple.
   8. If relationship triples greater than 1
      9. Loop over relationship triples
         10. If subject equals next triples subject
             and predicate not equals next triples predicate
                 and object equals next triples object
         End If
      End Loop
   End If
End Loop
End Loop
12. Loop over relationship triples to be updated array
13. Loop over type triples
   14. If relationships triples object equals type triples subject
       15. Create new type triple
       16. Update object of relationship triple with new triple subject
   End If
End Loop
End Loop

Listing 7.8: Algorithm for processing cycles
In case of SPARQL query shown in Listing 7.7, my algorithm in Listing 7.8 introduces a new concept c3 having the same type as that of c2 and then created a relationship from c1 to c3. The resulting SPARQL is shown below.

```
PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
SELECT DISTINCT ?c1 ?c2 ?c3 WHERE {
  ?c1 ns1:type ns2:ShallowWellbore.
  ?c2 ns1:type ns2:Field.
  ?c3 ns1:type ns2:Field.
  ?c1 ns2:shallowWellboreForField ?c2.
  ?c1 ns2:wellboreForField ?c3.
}
```

Listing 7.9: Type added to handle cycles

As we have seen, we must create two instances of same type since it is not possible to create more than one relationships between two concepts. There is requirement in OptiqueVQS that we should keep track of all the similar types in our query. In following SPARQL we can see that ShallowWellbore is related to Field concept by both shallowWellboreForField and wellboreForField relationships.

```
PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
SELECT DISTINCT ?c1 ?c2 ?c3 WHERE {
  ?c1 ns1:type ns2:ShallowWellbore.
  ?c2 ns1:type ns2:Field.
  ?c3 ns1:type ns2:Field.
  ?c1 ns2:shallowWellboreForField ?c2.
  ?c1 ns2:wellboreForField ?c3.
}
```

Listing 7.10: Similar types
To fulfill this requirement, an array of array is maintained where every inner array contains lists of variable names for a single concept that is instantiated more than once. This array is assigned to sameNodes property in our start node. In above case we will have following information.

```
"sameNodes": [
    [
        "?c2",
        "?c3"
    ]
]
```

Listing 7.11: Same nodes array

In addition to above information, I have also added a filter to JSON object saying which two concepts are equal.

```
FILTER(?c2 = ?c3)
```

Listing 7.12: Same types filter

This is done because our visual query follows a tree structures and not graph.
7.5 Validation of ontology

This thesis work aims to provide users a feature to edit or update a SPARQL query. It means users can type a new query, copy/paste an existing query and they can update a query generated from visual query. In all cases, it was necessary to check the query for syntax errors and validate it against underlying ontology. SPARQL parser has done syntax checking and I have manually implemented following validations.

7.5.1 Core concepts

Each node in our VQS diagram represents a variable with a type. A variable with a type represents a concept in our system. These concepts are also known as classes and they are defined in ontology. The first and most important thing to validate was these concepts. After parsing SPARQL query, I got a JSON object that contains all the information about query. I then fetched all the type-defining triples from this JSON object and created a list of them. All the core concepts in underlying ontology were retrieved via AJAX call to a web service. This list of types from SPARQL query was then verified against those concepts from ontology. If there were some types that were not present in ontology, error messages were generated for user and no further processing was done.

7.5.2 Concept fields

Once validation of core concepts was complete, I validated fields (properties) for these concepts. This was done by processing all the concepts one by one. Here again I made a service call to get all the properties for a concept from ontology. I then validated the properties in the query with properties from ontology. If there were any properties in query that were not found in property list returned from ontology, error messages were generated and no further processing was done.

7.5.3 Adjacencies

Adjacencies or relationships between concepts was the last thing to be validated. I needed to check if a relationship created in query between two concepts was valid according to ontology. For doing this, I made a service call to get all the neighboring concepts for a given concept from the ontology. If the relationship specified in query was found in the list of neighboring concepts it was a valid relationship otherwise error messages were generated for user and no further processing will be done.
7.6 Transformation of parsed JSON to Optique JSON

Once the SPARQL is parsed to a JSON object and that JSON object is pre-processed and validated the next phase is to transform it into OptiqueVQS JSON object. There are several steps involved in this transformation phase and each one of them is discussed in details in this section.

7.6.1 Start Node

When OptiqueVQS loads, it shows a start node as shown in figure below.

![Start node](image)

This node does not represent any concept in SPARQL and works as a starting point. After recent additions of new features, it now also contains configuration information about aggregates and similar concepts in the query. Since this node will always contain some standard information, it is first loaded as JSON from a static string. Following listing illustrates JSON representation of start node.

```json
{
    "id": "0",
    "data": {
        "id": "0",
        "label": "Untitled query",
        "desc": "Please provide a description here...",
        "icon": "",
        "$color": "#FF9900",
        "sequence": {},
        "aggregate": {},
        "isActive": "true"
    }
}
```
Listing 7.13: First node JSON

Label, description, icon and color are general properties to control the graphical representation in canvas. Details about how to process aggregates and sequence properties comes later in this section. The isActive property defines status of a node, when this property is true for a node, it means that node has the focus. Adjacencies specify relationships between nodes. In this case, start node is connected with first concept in SPARQL that has an id c1. The sameNodes property has been discussed in section 7.4.2.

7.6.2 Processing typed variables (Concepts)

While the start node has a special significance and behavior in OptiqueVQS, rest of all the nodes represent concepts in SPARQL query. For each of these nodes we need to process following areas.

1. Metadata
2. Outputs
3. Constraints
4. Adjacencies
**Metadata**

Metadata for a node contains detailed information about graphical representation in canvas. A metadata id attribute for the node is the URI of the concept with name space and concept name. An icon has the URL of the icon to be shown as a background image for the node. Every node has a color property, which is different for active and inactive concepts. Only one node can be active at one time. Since it is difficult to keep track of active and inactive nodes in a SPARQL query, I will always make the start node as active node while switching from textual mode to graphical mode. Listing 7.14 illustrates the metadata information of a type variable.

```
"id": "http://sws.ifi.uio.no/vocab/npd-v2#Wellbore",
"name": "Wellbore",
"label": "Wellbore",
"ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
"icon": "../../icons/Wellbore.png",
"hId": "Wellbore_c1",
"$color": "#FFFFFF",
```

Listing 7.14: Metadata for concept

The SPARQL parser gives us the type/concept triples in following format.

```
{
  "subject": "?c1",
  "predicate": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
  "object": "http://sws.ifi.uio.no/vocab/npd-v2#Wellbore"
}
```

Listing 7.15: Type triple

Object property in above listing represents the metadata id of the concept in Optique JSON. The name, label and namespace are extracted from object property by separating name space and the concept name.
Outputs

For every concept, there can be none or many outputs in a SPARQL query. While processing a concept we must take care of all these outputs. To accomplish this I get all the triples where current concept is the subject and object is neither a type nor a type variable. Consider following listing where we have a concept Wellbore with three outputs. The outputs are name, wellboreMaxInclation and wellboreDrillingDays.

```sparql
PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
PREFIX ns3: <http://www.optique-project.eu/well-ontology/>

SELECT DISTINCT ?c1 ?a1 ?a2 ?a3 WHERE {
    ?c1 ns1:type ns2:Wellbore.
    ?c1 ns3:name ?a1.
}
```

Listing 7.16: SPARQL outputs

An object for each of these outputs has been created with metadata about the output. All of these outputs are added to an array for the current node. Id for the output metadata is fetched from the parsed JSON object. Name/label and namespace (ns) are extracted from the URI by separating output attribute and name space. Property ald is a variable name and is generated with an integer postfix whereas hId is a combination of output name and variable name.

```json
"output": [
{
    "id": "http://www.optique-project.eu/well-ontology/name",
    "name": "name",
    "ns": "http://www.optique-project.eu/well-ontology/",
    "label": "name",
    "type": "string",
}
```
"aId": "a1",
"hId": "name_a1"
},
{
"id": "http://sws.ifi.uio.no/vocab/npd-v2#wellboreMaxInclation",
"name": "wellboreMaxInclation",
"ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
"label": "wellboreMaxInclation",
"type": "string",
"aId": "a2",
"hId": "wellboreMaxInclation_a2"
},
{
"id": "http://sws.ifi.uio.no/vocab/npd-v2#wellboreDrillingDays",
"name": "wellboreDrillingDays",
"ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
"label": "wellboreDrillingDays",
"type": "string",
"aId": "a3",
"hId": "wellboreDrillingDays_a3"
}
]

Listing 7.17: JSON outputs

Constraints

Every node can have constraints on one or many of its outputs. Constraints are meant to filter the information returned by the SPARQL query. Let suppose we have following SPARQL with a simple filter that will give us all the awards where award name is “MyAward”. 

45
Our SPARQL parser will parse the above query with following information about filter.

```
{
    "type": "filter",
    "expression": {
        "type": "operation",
        "operator": "regex",
        "args": [
            "?a1",
            "\"MyAward\""
        ]
    }
}
```

Listing 7.19: SPARQL Filter in JSON

In case there are many filters in SPARQL query, there will be an array of filters specifying details about each of them. While processing a concept I have only processed the filters on the outputs of that particular concept. Filters can be simple like string comparisons and they can complex with more than one logical operations. They are processed according to their complexity levels. Once all the filters for a concept are processed they are assigned to constraints property in JSON.
Here is an example of simple string comparison filter in Optique JSON.

```
"constraint": [
  {
    "id": "http://www.optique-project.eu/well-ontology/name",
    "name": "name",
    "ns": "http://www.optique-project.eu/well-ontology/",
    "label": "name",
    "type": "string",
    "constr": "MyAward",
    "constrType": "eq",
    "aId": "a1"
  }
]
```

Listing 7.20: String comparison filter

**Relationships**

The OptiqueVQS is built with left to right hierarchical tree structure. Nodes represent the concepts and they are connected with each other by relationships also known as adjacencies. When converting a SPARQL query to OptiqueVQS these relationships are processed while processing concepts. For each concept, I fetch all the relationships and this is done by getting all the triples where current concept is subject and object is a concept. In following SPARQL query there are two concepts Wellbore and Field. These two are related by a property WellboreForField.

```
PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
SELECT ?c1 ?a1 ?c2 ?a2 WHERE {
  ?c1 ns1:type ns2:Wellbore.
  ?c2 ns1:type ns2:Field.
  ?c1 ns2:WellboreForField ?c2.
```
In our parsed JSON there is a relationship triple created to show this relationship.

```
{
  "subject": "?c1",
  "predicate": "http://sws.io.no/vocab/npd-v2#WellboreForField",
  "object": "?c2"
}
```

Listing 7.22: SPARQL adjacencies RJSON

From above triple, I have added following information for node c1 to our OptiqueVQS JSON about relationship between c1 and c2.

```
"adjacencies": [
  {
    "nodeTo": "c2",
    "data": {
      "id": "http://sws.uio.no/vocab/npd-v2#WellboreForField",
      "name": "WellboreForField",
      "label": "WellboreForField",
      "ns": "http://sws.ifi.uio.no/vocab/npd-v2#"
    }
  }
]
```

Listing 7.23: SPARQL adjacencies Optique JSON
Inverse Relationships

In SPARQL, we can have Inverse properties between two types and they are implemented in OptiqueQS as well. An inverse property reverses the relationship direction and swaps the role of subject and object in a triple [29]. Here is an example SPARQL query with inverse relationship.

```
PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
PREFIX ns3: <http://www.optique-project.eu/well-ontology/>
SELECT ?c1 ?a1 ?c2 ?a2 WHERE {
  ?c1 ns1:type ns2:Company.
  ?c2 ns1:type ns2:Field.
  ?c1 ns3:name ?a1.
}
```

Listing 7.24: SPARQL with inverse properties

Parsed JSON has following information about inverse relationship specified in above query.

```
{
  "subject": "?c1",
  "predicate": {
    "type": "path",
    "pathType": "^",
    "items": [
      "http://sws.ifi.uio.no/vocab/npd-v2#currentFieldOperator"
    ]
  },
  "object": "?c2"
}
```

Listing 7.25: RJSON with inverse properties
Above information is processed as a special case while building adjacencies and following information is added to OptiqueVQS JSON.

"adjacencies": [
  {
    "nodeTo": "c2",
    "data": {
      "id": "http://url/nd-v2#currentFieldOperator_inverseProp",
      "name": "^currentFieldOperator",
      "label": "(inv) currentFieldOperator",
      "ns": "http://sws.ifi.uio.no/vocab/npd-v2#"
    }
  }
]

Listing 7.26: Optique JSON with Inverse properties

7.6.3 Aggregates

Aggregate functions such as AVG, MAX, MIN, SUM and COUNT are applied on group of solutions. In OptiqueVQS, full specification of SPARQL aggregates is under development. Simple aggregates functionality however, has been implemented. In our context, an aggregate function is supposed to be applied on a node output and we have to group all the other outputs accordingly. This means aggregate function will not be related to a particular node but it will be applied on the query level.

In following SPARQL query, I have a COUNT on dimension names and it is in ascending order.

```sparql
PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
PREFIX ns3: <http://www.optique-project.eu/well-ontology/>
SELECT ?c1 ?a1 ?c2 (COUNT(?a2) AS ?COUNT_a2) WHERE {
  ?c1 ns1:type ns2:Company.
  ?c2 ns1:type ns3:Dimension.
} ORDER BY ?COUNT_a2
```
When above query is parsed I get following information about aggregate COUNT on a2.

```
"variables": [
    "?c1",
    "?a1",
    "?c2",
    {
      "expression": {
        "expression": "?a2",
        "type": "aggregate",
        "aggregation": "count",
        "distinct": false
      },
      "variable": "?COUNT_a2"
    }
]
```

This information about aggregates functions is added to the first node of the OptiqueVQS JSON. Sequence property has the information about sorting, which is sort up on COUNT_a2 while aggregate has details about aggregate function. In this case, we are taking count on a2 and count variable is COUNT_a2.

```
"sequence": {
    "op": "sort_up",
```

Listing 7.27: SPARQL with aggregate

Listing 7.28: Aggregates in RJSON
"vr": "COUNT_a2"
},
"aggregate": {
    "op": "take_count",
    "vr": "a2",
    "vra": "COUNT_a2",
    "vrha": "count_name_a2"
}

Listing 7.29: Start node with aggregate information.
Chapter 8

Alternative approach for transformation

I have discussed the transformation of parsed JSON to OptiqueVQS JSON in chapter 7. An alternative approach was also explored for generating visual query from parsed JSON. This approach was about directly manipulating a tree object that is responsible for creating visual elements in OptiqueVQS. In this approach, the JSON object from SPARQL parser was transformed into another temporary JSON object. That temporary JSON object contains an array of nodes and a single node has following structure.

```json
{
    "type": "conceptSelected",
    "content":{
        "concept": {
            "id": "http://sws.ifi.uio.no/vocab/npd-v2#Company",
            "name": "Company",
            "label": "Company",
            "ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
            "icon": "../../icons/Company.png"
        },
        "property": {
            "id": "http://www.optique.eu/well-ontol/hasDimension",
            "name": "hasDimension",
            "label": "hasDimension",
            "ns": "http://www.optique.eu/well-ontology/"
        }
    },
    "attributes": [],
    "constraints": []
}
```

Listing 8.1: Text to tree JSON structure
The type property for a node decides whether it is an active or a passive node. Content has two sub properties concept and property. Concept property has information about the graphical representation of the concept while property has the information about the relationship to next node. Attributes and constraints arrays have the information about the outputs and constraints on the current node.

While building visual query from this JSON object all the nodes in the array are iterated over and respective methods for tree object are called as shown in Listing 8.2. First, a node is added to tree then attributes, constraints and relationships are added for that node.

```javascript
  tree.addNode('', data.content, 'stack', 'active');
  tree.addAttr(attrForCurrentConcept[a], 'stack');
  tree.addConstraint(constForCurrentConcept[c], 'stack');
```

Listing 8.2: Text to tree calling tree methods in OptiqueVQS.

There are pros and cons of both approaches. One of the benefit of using this approach was that we do not have to worry about number postfixes for class and constraint variables as we are directly manipulating the tree. The problem however is that when user will go from text to visual mode, he will see all the concepts loading while tree is being built. This can be quite annoying once we have large number of concepts, outputs and constraints in our SPARQL. Dealing with aggregates is another problem since aggregates are applied on query level and not individual concepts.
Part III

III Conclusion
Chapter 9

Summary

Chapter 1 discusses the background and introduction of this thesis work. Introduction part discusses the importance and benefits of using visual query systems. It also discusses how such systems can help in removing the major bottlenecks in end-user access to big data. The problem statements describe limitations of OptiqueVQS and usefulness of collaborative query formulation. A brief description of Optique project and its architecture has been discussed in Optique section. An introduction to a UI mashups and how this approach is applied in OptiqueVQS comes next.

Chapter 2 gives a brief description of major semantic technologies that are included in scope of this thesis work such as RDF, RDFS, Ontologies and SPARQL.

Chapter 3 discusses the core concept of collaborative query formulation and its types such as active and passive collaboration.

Chapter 4 summarize the scientific work that is related to this thesis work including different kind of code engineering solutions, visual query systems etc.

Chapter 5 describes the architecture of this thesis work.

Chapter 6 discusses the tools and technologies used for development of this thesis work such as SPARQL parser, InfoVis Toolkit and jQuery mobile.

Chapter 7 explains the implementation of this thesis work with help of examples. It includes all the details of implementation such as making SPARQL editable, cleaning SPARQL, parsing SPARQL, pre-processing, validating and at the end transforming queries to visual format.

Chapter 8 discusses an alternative approach to generate visual queries in OptiqueVQS

Chapter 10 evaluates and discuss the overall progress, achievements and limitations of this work.

Chapter 11 concludes the work accomplished and future improvements.
Chapter 10

Discussion and evaluation

The goal of this thesis work was to implement a collaborative query formulation solution that enables domain specialists and technical experts to formulate queries in collaboration to extract information from large data sources. The starting point was an already implemented prototype OptiqueVQS that applies ontology-based data access approach to access big data. In OptiqueVQS, domain concepts and relationships are represented by visual elements and end-users can create visual queries by manipulating these visual elements. The system then converts visual queries into SPARQL queries for retrieving information from underlying data sources.

This thesis work has added a new feature that allows creating and editing SPARQL queries. This feature also make it possible to convert SPARQL queries to visual queries that allows query formulation in both textual and visual modes.

OptiqueVQS is an ongoing project and currently it only supports special kind of SELECT queries with a certain amount of complexity. Currently SPARQL to visual query transformation supports exactly the same complexity level. Any new functionality in visual queries will require updating the logic for transformation from SPARQL to visual queries.

I came across some limitations during implementation of this work that are worth mentioning here. One of these limitations came while generating SPARQL queries from visual queries and it was about how to retain the complete graphical information from visual query in SPARQL. This information includes node color, status being active or inactive, node icons etc. I have implemented a simple functionality with the consent from supervisors. I will always set first node as active when transforming a SPARQL query to visual query while icon URL will be build assuming that icon name is same as concept name. As a future improvement, this graphical information can be specified as comments in SPARQL query when generating it from visual query, in this way the exact information can be retained while switching between two modes.

Another limitation in OptiqueVQS is its support for only a certain type of SELECT queries. The reason for this is to keep things simple for end-users in start. As the users will get familiar with system, more features can be added according to their requirements.
There has been a parallel thesis work recently completed with title “OptiqueNLQF\(^1\): A natural language query formulation system based on Semantic Technologies” by Tomas Stein Sæbu. OptiqueNLQF is capable of handling simple natural language phrases for generating SPARQL queries. As a future work the functionality implemented as part of current thesis and the one developed in OptiqueNLQF can be combined together to develop a system that allows transformation from natural language into SPARQL and from SPARQL into visual query. Such a system will enable users with different expertise and background to collaborate with each other for formulations of queries to achieve common goals.

\(^1\) [http://www.mn.uio.no/ifi/studier/masteroppgaver/ltg/natural-language-query-formulation.html](http://www.mn.uio.no/ifi/studier/masteroppgaver/ltg/natural-language-query-formulation.html)
Chapter 11

Conclusion

As a part of this thesis work, I have implemented a solution where domain experts and technical experts can collaborate in a visual query system to formulate queries for accomplishing common goals. My focus has been OptiqueVQS where end-users can formulate visual queries to extract information from huge data sources according to their needs. The challenge was to make a collaborative solution where technical experts can help end-users if they are struggling to get desired results from their visual queries. This would have been easier to accomplish if a functionality for generating visual queries from textual queries would have been possible because technical users have better understanding of textual queries, in this case SPARQL.

A SPARQL query is only a string a characters but it follows a certain grammar that make it possible to parse it. A number of different parsing strategies and solutions were considered and finally SPARQL.js was chosen for reasons such as client side implementation in JavaScript, support for full specification of SPARQL 1.1 and regular maintenance by its author. This parser also ensures that SPARQL query is syntactically correct and display appropriate errors if this is not the case.

Once the SPARQL parsing was in place, the next step was to validate the SPARQL against underlying ontology. This was accomplished by making AJAX calls to a service that returns ontology information such as concepts in ontology, concept properties and relationships.

After validation remaining work was about processing all the minor details from parsed JSON object and building the visual query. Two main approaches were applied for this task. The first one was to map all the information from parsed JSON to another JSON object. This JSON object was then processed to directly manipulate the tree in OptiqueVQS calling its different methods.

The second approach was about transforming parsed JSON object to OptiqueVQS JSON object and loading the visual query from it. The second approach was preferred as the complexity of VQS is increasing and it was found to be more efficient and logical.
11.1 Future Work

OptiqueVQS is currently an ongoing project and new features are continuously being added. The query formulation in getting richer and complexity is increasing. For example, recently aggregates have been added and logic for handling them have been implemented as a part of this thesis work. There will be many new features coming in near future and SPARQL to visual query part will keep on evolving together with main features in the system.

In general, there are almost no other solutions where collaborative query formulation is possible in semantic web domain. It is a good start and solution like this can be generalized to behave like a middle layer between textual SPARQL and any other visual query system.
Bibliography


*SPARQL 1.1 Query Language*. 2013; Available from: http://www.w3.org/TR/sparql11-query/.

Part IV

IV Appendices
Appendix A

This thesis work is implemented as a part of OptiqueVQS. Currently this work is not available online as it needs approval from different actors involved in Optique project. A demo of OptiqueVQS is however available from following links on fluidops website.

- http://fact-pages.fluidops.net/interface/index.html
- http://fact-pages.fluidops.net/resource/VisualQueryFormulationNPD

Latest versions of Chrome and Firefox browsers are recommended for using OptiqueVQS.
Appendix B

In this appendix, I have included four examples with all four representations. Four representations are shown in following order.

1. Visual query
2. SPARQL query
3. Parsed JSON
4. Optique JSON

14.1 Simple query with two concepts

```
PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
SELECT ?c1 ?a1 ?a4 ?c2 ?a2 ?a3 WHERE {
  ?c1 ns1:type ns2:Wellbore.
  ?c2 ns1:type ns2:Field.
  ?c1 ns2:developmentWellboreForField ?c2.
  ?c1 ns2:name ?a1.
  ?c1 ns2:wellboreWellType ?a4.
}
```

SPARQL
{  
  "type": "query",
  "queryType": "SELECT",
  "variables": [
    "?c1",
    "?a1",
    "?a4",
    "?c2",
    "?a2",
    "?a3"
  ],
  "where": [
    {
      "type": "bgp",
      "triples": [
        {
          "subject": "?c1",
          "predicate": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
          "object": "http://sws.ifi.uio.no/vocab/npd-v2#Wellbore"
        },
        {
          "subject": "?c2",
          "predicate": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
          "object": "http://sws.ifi.uio.no/vocab/npd-v2#Field"
        },
        {
          "subject": "?c1",
          "predicate": "http://sws.ifi.uio.no/vocab/npd-
v2#developmentWellboreForField",
    "object": "?c2"
},
{
    "subject": "?c1",
    "predicate": "http://sws.ifi.uio.no/vocab/npd-v2#name",
    "object": "?a1"
},
{
    "subject": "?c1",
    "predicate": "http://sws.ifi.uio.no/vocab/npd-v2#wellboreWellType",
    "object": "?a4"
},
{
    "subject": "?c2",
    "predicate": "http://sws.ifi.uio.no/vocab/npd-v2#name",
    "object": "?a2"
},
{
    "subject": "?c2",
    "predicate": "http://sws.ifi.uio.no/vocab/npd-v2#status",
    "object": "?a3"
}
]
}
],
"prefixes": {
    "ns1": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
    "ns2": "http://sws.ifi.uio.no/vocab/npd-v2#"
[{
  "id": "0",
  "data": {
    "id": "0",
    "label": "Untitled query",
    "desc": "Please provide a description here...",
    "icon": "",
    "$color": "#FF9900",
    "sequence": {},
    "aggregate": {},
    "isActive": "true"
  },
  "adjacencies": [
    {
      "nodeTo": "c1",
      "data": {
        "id": "",
        "name": "",
        "label": "",
        "ns": ""
      }
    }
  ],
  "sameNodes": []
},]
{
    "id": "c1",
    "data": {
        "id": "http://sws.ifi.uio.no/vocab/npd-v2#Wellbore",
        "name": "Wellbore",
        "label": "Wellbore",
        "ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
        "icon": "../../icons/Wellbore.png",
        "hId": "Wellbore_c1",
        "$color": "#FFFFFF",
        "output": [
            {
                "id": "http://sws.ifi.uio.no/vocab/npd-v2#name",
                "name": "name",
                "ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
                "label": "name",
                "type": "string",
                "aId": "a1",
                "hId": "name_a1"
            },
            {
                "id": "http://sws.ifi.uio.no/vocab/npd-v2#wellboreWellType",
                "name": "wellboreWellType",
                "ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
                "label": "wellboreWellType",
                "type": "string",
                "aId": "a4",
                "hId": "wellboreWellType_a4"
            }
        ]
    }
}


},
"constraint": []
},
"adjacencies": [
{
"nodeTo": "c2",
"data": {
"id": "http://sws.ifni.uio.no/vocab/npd-v2#developmentWellboreForField",
"name": "developmentWellboreForField",
"label": "developmentWellboreForField",
"ns": "http://sws.ifni.uio.no/vocab/npd-v2#"
}
}
]
,
{
"id": "c2",
"data": {
"id": "http://sws.ifni.uio.no/vocab/npd-v2#Field",
"name": "Field",
"label": "Field",
"ns": "http://sws.ifni.uio.no/vocab/npd-v2#",
"icon": "../icons/Field.png",
"hId": "Field_c2",
"$color": "#FFFFFF",
"output": [
{
"id": "http://sws.ifni.uio.no/vocab/npd-v2#name",
"name": "name",
}]
}
Optique JSON

14.2 Query with two concepts and one filter

Visual query
PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>

SELECT ?c1 ?a1 ?a2 ?c2 ?a3 ?a4 WHERE {
    ?c1 ns1:type ns2:Wellbore.
    ?c2 ns1:type ns2:Field.
    ?c1 ns2:developmentWellboreForField ?c2.
    ?c1 ns2:name ?a1.
    ?c1 ns2:wellboreWellType ?a2.
    ?c2 ns2:name ?a3.
    FILTER(regex(?a2, "SHALLOW", "i").
}

SPARQL

{
  "type": "query",
  "queryType": "SELECT",
  "variables": [
    "?c1",
    "?a1",
    "?a2",
    "?c2",
    "?a3",
    "?a4"
  ],
  "where": [
    {
      "type": "bgp",
      "triples": [
        {
          "type": "query",
          "queryType": "SELECT",
          "variables": ["?c1", "?a1", "?a2", "?c2", "?a3", "?a4"]
        }
      ]
    }
  ]
}
"subject": "?c1",
"predicate": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
"object": "http://sws.ifi.uio.no/vocab/npd-v2#Wellbore"
},
{
"subject": "?c2",
"predicate": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
"object": "http://sws.ifi.uio.no/vocab/npd-v2#Field"
},
{
"subject": "?c1",
"predicate": "http://sws.ifi.uio.no/vocab/npd-v2#developmentWellboreForField",
"object": "?c2"
},
{
"subject": "?c1",
"predicate": "http://sws.ifi.uio.no/vocab/npd-v2#name",
"object": "?a1"
},
{
"subject": "?c1",
"predicate": "http://sws.ifi.uio.no/vocab/npd-v2#wellboreWellType",
"object": "?a2"
},
{
"subject": "?c2",
"predicate": "http://sws.ifi.uio.no/vocab/npd-v2#..."
"predicate": "http://sws.ifi.uio.no/vocab/npd-v2#name",
"object": "?a3"
},
{
"subject": "?c2",
"predicate": "http://sws.ifi.uio.no/vocab/npd-v2#status",
"object": "?a4"
}
]
},
{
"type": "filter",
"expression": {
"type": "operation",
"operator": "regex",
"args": [
"?a2",
"\"SHALLOW\"",
"\"i\""
]
}
}
]
,"prefixes": {
"ns1": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
"ns2": "http://sws.ifi.uio.no/vocab/npd-v2#"
}
}
"name": "Wellbore",
"label": "Wellbore",
"ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
"icon": "../../icons/Wellbore.png",
"hId": "Wellbore_c1",
"$color": "#FFFFFF",
"output": [
{
    "id": "http://sws.ifi.uio.no/vocab/npd-v2#name",
    "name": "name",
    "ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
    "label": "name",
    "type": "string",
    "aId": "a1",
    "hId": "name_a1"
},
{
    "id": "http://sws.ifi.uio.no/vocab/npd-v2#wellboreWellType",
    "name": "wellboreWellType",
    "ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
    "label": "wellboreWellType",
    "type": "string",
    "aId": "a2",
    "hId": "wellboreWellType_a2"
}
],
"constraint": [
{
    "id": "http://sws.ifi.uio.no/vocab/npd-v2#constraint"
}]}
14.3 Query with inverse relationship

PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
PREFIX ns3: <http://www.optique-project.eu/well-ontology/>

SELECT ?c1 ?a1 ?c2 ?a2 ?a3 WHERE {
  ?c1 ns1:type ns2:Company.
  ?c2 ns1:type ns2:Field.
  ?c1 ns3:name ?a1.
  ?c2 ns2:supplyBase ?a3.
}

{  
  "type": "query",
  "queryType": "SELECT",
  "variables": [
    "?c1",
    "?a1",
    "?c2",
    "?a2",
    "?a3"
  ]
}
"where": [
  {
    "type": "bgp",
    "triples": [
      {
        "subject": "?c1",
        "predicate": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
        "object": "http://sws.ifi.uio.no/vocab/npd-v2#Company"
      },
      {
        "subject": "?c2",
        "predicate": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
        "object": "http://sws.ifi.uio.no/vocab/npd-v2#Field"
      },
      {
        "subject": "?c1",
        "predicate": {
          "type": "path",
          "pathType": "^",
          "items": [
            "http://sws.ifi.uio.no/vocab/npd-v2#currentFieldOperator"
          ]
        },
        "object": "?c2"
      },
      {

"subject": "?c1",
"predicate": "http://www.optique-project.eu/well-ontology/name",
"object": "?a1"
},
{
  "subject": "?c2",
  "predicate": "http://sws.ifi.uio.no/vocab/npd-v2#name",
  "object": "?a2"
},
{
  "subject": "?c2",
  "predicate": "http://sws.ifi.uio.no/vocab/npd-v2#supplyBase",
  "object": "?a3"
}
]

"prefixes": {
  "ns1": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
  "ns2": "http://sws.ifi.uio.no/vocab/npd-v2#",
  "ns3": "http://www.optique-project.eu/well-ontology/"
}
}

Parsed JSON

[
  {
    "id": "0",

"data": {
    "id": "0",
    "label": "Untitled query",
    "desc": "Please provide a description here...",
    "icon": "",
    "$color": "#FF9900",
    "sequence": {},
    "aggregate": {},
    "isActive": "true"
},

"adjacencies": [
    {
        "nodeTo": "c1",
        "data": {
            "id": "",
            "name": "",
            "label": "",
            "ns": ""
        }
    }
],

"sameNodes": []
},

{
    "id": "c1",
    "data": {
        "id": "http://sws.ifi.uio.no/vocab/npd-v2#Company",
        "name": "Company",
        "label": "Company",
        "ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
    }
}
"icon": ".../icons/Company.png",
"hId": "Company_c1",
"$color": "#FFFFFF",
"output": [
  {
    "id": "http://www.optique-project.eu/well-ontology/name",
    "name": "name",
    "ns": "http://www.optique-project.eu/well-ontology/",
    "label": "name",
    "type": "string",
    "aId": "a1",
    "hId": "name_a1"
  }
],
"constraint": []
],
"adjacencies": [
  {
    "nodeTo": "c2",
    "data": {
      "id": "http://sws.ifi.uio.no/vocab/npd-v2#currentFieldOperator_inverseProp",
      "name": "^currentFieldOperator",
      "label": "(inv) currentFieldOperator",
      "ns": "http://sws.ifi.uio.no/vocab/npd-v2#"
    }
  }
]
"id": "c2",
"data": {
  "id": "http://sws.ifi.uio.no/vocab/npd-v2#Field",
  "name": "Field",
  "label": "Field",
  "ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
  "icon": "../../icons/Field.png",
  "hId": "Field_c2",
  "$color": "#FFFFFF",
  "output": [
    {
      "id": "http://sws.ifi.uio.no/vocab/npd-v2#name",
      "name": "name",
      "ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
      "label": "name",
      "type": "string",
      "aId": "a2",
      "hId": "name_a2"
    },
    {
      "id": "http://sws.ifi.uio.no/vocab/npd-v2#supplyBase",
      "name": "supplyBase",
      "ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
      "label": "supplyBase",
      "type": "string",
      "aId": "a3",
      "hId": "supplyBase_a3"
    }
  ],
  "constraint": []
}
14.4 Query with aggregates

PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
SELECT ?c1 ?a1 ?a3 ?a2 (COUNT(?c2) AS ?COUNT_c2) WHERE {
   ?c1 ns1:type ns2:Wellbore.
   ?c2 ns1:type ns2:Field.
   ?c1 ns2:explorationWellboreForField ?c2.
   ?c1 ns2:wellboreWellType ?a1.
   ?c1 ns2:name ?a3.
} GROUP BY ?c1 ?a1 ?a3 ?a2
{  
  "type": "query",
  "queryType": "SELECT",
  "variables": [
    "?c1",
    "?a1",
    "?a3",
    "?a2",
    {
      "expression": {
        "expression": "?c2",
        "type": "aggregate",
        "aggregation": "count",
        "distinct": false
      },
      "variable": "?COUNT_c2"
    }
  ],
  "where": [
    {
      "type": "bgp",
      "triples": [
        {
          "subject": "?c1",
          "predicate": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
          "object": "http://sws.ifi.uio.no/vocab/npd-v2#Wellbore"
        }
      ]
    }
  ]
}
"group": [
  {
    "expression": "?c1"
  },
  {
    "expression": "?a1"
  },
  {
    "expression": "?a3"
  },
  {
    "expression": "?a2"
  }
],
"prefixes": {
  "ns1": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
  "ns2": "http://sws.ifi.uio.no/vocab/npd-v2#"
}
}
"id": "http://sws.ifi.uio.no/vocab/npd-v2#explorationWellboreForField",
"name": "explorationWellboreForField",
"label": "explorationWellboreForField",
"ns": "http://sws.ifi.uio.no/vocab/npd-v2#"
}

]
}

{
"id": "c2",
"data": {
"id": "http://sws.ifi.uio.no/vocab/npd-v2#Field",
"name": "Field",
"label": "Field",
"ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
"icon": "../../icons/Field.png",
"hId": "Field_c2",
"$color": "#FFFFFF",
"output": [
{
"id": "http://sws.ifi.uio.no/vocab/npd-v2#name",
"name": "name",
"ns": "http://sws.ifi.uio.no/vocab/npd-v2#",
"label": "name",
"type": "string",
"aId": "a2",
"hId": "name_a2"
}
],
"constraint": []
},
"adjacencies": []
}
]

Optique JSON