Ontology-Based SPARQL Extension Ranker
Basic Implementation in the Context of OptiqueVQS and Comparison with Collaborative Filtering
Magnus Arneberg Nilsen
Master’s Thesis, Autumn 2017
This master’s thesis is submitted under the master’s program *Informatics: Programming and Networks*, at the Department of Informatics, University of Oslo. The scope of the thesis is 60 credits.
Abstract

This paper addresses the basic implementation and possible competitiveness of an ontology-based ranking method, that for a SPARQL query under construction, generates a ranking of likely extensions, based on a set of past queries. An extension’s rank is its conditional probability of being seen in the set of past queries, where the condition is that its seen on a query with the same constraints up to the extension point. The context here is SPARQL queries constructed with the visual query system OptiqueVQS, so the types of extensions is limited to the ones it can make. A ranking method is needed to let end-users of OptiqueVQS faster locate wanted extensions, because the number of possible extensions gets quickly hard to manage relative to the size of the ontology used. OptiqueVQS is part of the EU project Optique - Scalable End-user Access to Big Data. Optique uses the paradigm of ontology-based data access (OBDA), empowering end-users with the ability to query after data in domain vocabulary and relations that they are familiar with. How much of an advantage would it be to use an ontology-based ranker in this ontological setting? Further enhanced versions of the ranker, can in different ways take the semantics in to an account to bias the rankings in ways other methods do not. To get an idea of the possible competitiveness of future enhanced versions, a collaborative filtering-based ranking method is implemented to compare against. These two methods are pitted against each other in an experiment where they are scored based on how high up the rankings certain intentionally removed extensions get.
Acknowledgments

I would like to thank my supervisors Ahmet Soylu and Martin Giese, the logic and intelligent data (LogID) group, the students and faculty at the institute of informatics, and all of the good people at the University of Oslo. I am grateful to all that contribute and have contributed, in all and every way, to open source projects. The projects that are used directly in the implementations here, are Apache-Jena, RDF4J, and Apache-Mahout. They are nice. This opportunity would not be without the EU project Optique, and the intellectual cooperation between many universities, corporations and organizations across countries, that has gone into it. Also would like to thank Dag Langmyhr and Martin Helsø for their online LaTeX resources.

OptiqueVQS is funded by the FP7-ICT of the European Commission (EU/EC) under Grant Agreement 318338, “Optique”.

This work was conducted using the Protégé resource, which is supported by grant GM10331601 from the National Institute of General Medical Sciences of the United States National Institutes of Health.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>ii</td>
</tr>
<tr>
<td>Contents</td>
<td>iii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>v</td>
</tr>
<tr>
<td><strong>I Preliminaries</strong></td>
<td>1</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>2</td>
</tr>
<tr>
<td>1.1 Optique</td>
<td>3</td>
</tr>
<tr>
<td>1.2 OptiqueVQS</td>
<td>7</td>
</tr>
<tr>
<td>1.3 Ontology-based Ranking</td>
<td>11</td>
</tr>
<tr>
<td>2 Foundation</td>
<td>18</td>
</tr>
<tr>
<td>2.1 RDF</td>
<td>19</td>
</tr>
<tr>
<td>2.2 RDFS</td>
<td>20</td>
</tr>
<tr>
<td>2.3 OWL.2</td>
<td>21</td>
</tr>
<tr>
<td>2.4 TURTLE</td>
<td>23</td>
</tr>
<tr>
<td>2.5 SPARQL</td>
<td>23</td>
</tr>
<tr>
<td>2.6 Jena</td>
<td>25</td>
</tr>
<tr>
<td>2.7 RDF4J</td>
<td>28</td>
</tr>
<tr>
<td>2.8 Mahout</td>
<td>34</td>
</tr>
<tr>
<td>2.9 Protégé</td>
<td>35</td>
</tr>
<tr>
<td>2.10 Others</td>
<td>39</td>
</tr>
<tr>
<td><strong>II Ranking Methods</strong></td>
<td>40</td>
</tr>
<tr>
<td>3 Ontology-based Ranker</td>
<td>41</td>
</tr>
<tr>
<td>3.1 Modification</td>
<td>44</td>
</tr>
<tr>
<td>3.2 QueryLog extends LinkedHashMap&lt;Path, Query&gt;</td>
<td>50</td>
</tr>
<tr>
<td>3.3 LogModel extends ModelCom</td>
<td>52</td>
</tr>
</tbody>
</table>
List of Tables

1.1 Optique’s key features .......................... 4
1.2 Ontology-based Ranking Steps .................. 16
2.1 SPARQL Query Forms .......................... 25
2.2 Query Times in Milliseconds .................. 33
PART I

Preliminaries
CHAPTER 1

Introduction

In this thesis, the implementation of the basic ontology-based ranking method, builds upon the strategy laid out in the paper ‘Towards Exploiting Query History for Adaptive Ontology-Based Visual Query Formulation’. 

I try to explore more of the concept and test the competitiveness of the ranking method, by looking at it relative to a ranking method implemented to use collaborative filtering as its strategy to better suggest extensions to end-users constructing their queries with OptiqueVQS.

On the structure of this thesis. It is divided into three parts. teh preliminaries part, introduces the context and background, Optique, its visual query system OptiqueVQS, and the formal problem. Furthermore, the preliminaries explains the fundamental technologies used, why they are used and for what they are used for. The ranking methods part deals with implementation of the methods, and the problems that was encountered. The last part is the evaluation of the ranking methods. This evaluation part deals with the details of the experiment done, what results it produced, what to take from the results and some elaboration on further improved ranking methods.

Four whom is this paper valuable to read? Well that would be the person who wishes to implement an enhanced version of the basic ontology-based ranker presented here. Who's implementation might be integrated with OptiqueVQS. The time for that would be when OptiqueVQS has a stable API to program against, and Optique has some real end-user made queries and use-cases to program for. The ranker implementation that eventually will be used in the project, will have had to be implemented with a clear picture of the query preprocessing too take place and the data structures to be sent between components, in order to be optimized and functional.

The implementation details that are presented, shows how to solve the general problems and highlights general functionality any ranker of this type needs. All this is to help the next person with their implementation, to make it easier to implement the enhanced versions, to smooth out the basic trouble areas. The details specific to this basic ranking method is not that important, better design choices can be made in the future, with end-user data and with the lessens learned here. So the purpose of this paper is to provide a bootstrap into the subject matter, the technologies to consider and how they can be used.
1.1 Optique

Optique’s mission is to make big data directly accessible by end-users in a way that is scalable and agile. The Optique project is coordinated in Norway. University of Oslo is part of the Optique consortium. My two supervisors are: professor Martin Giese and postdoctoral research fellow Ahmet Soylu, both at University of Oslo. Martin Giese is the Assistant Scientific Director and oversees the work that goes into Optique from University of Oslo. Ahmet Soylu is in charge of Optique’s visual query system OptiqueVQS, the component that this paper tries to help along.

Companies like Siemens AG and Statoil ASA, both in the Optique Consortium, have large amounts of data in their respective domains. Maximally exploiting such data is increasingly critical to competitiveness. An example ontology as could be used by Statoil is used throughout this paper, it contains such concepts as “well” and “wellbore”. This ontology and accompanying data set, available on the Optique platform, is as would be used to model data from the exploration of new possible wells in the off-shore oil industry.

Unfortunately exploiting available data to the fullest becomes rather hard to do, due to the explosion in the size and complexity of the data sets, from for example actives like wellbore exploration. The engineers of these companies, need specific relevant data from the data sets to do their job, with big data they often end up using their time querying ineffectively, or getting IT experts involved that then have to learn about the domain to understand what the engineers are after.

The existing approaches to tackle the bottleneck of time spend searching for data, by limiting data access to a restricted set of predefined queries. This is the approach taken in simple cases, where the queries can be predefined and the data sources are uniform. Optique’s approach on the other hand, lets the engineer construct queries in terms of known domain concepts and relations. This taking advantage of the engineer’s in depth knowledge of the domain, and lets Optique create a query optimized for how the data repositories underneath are organized.

![Figure 1.1: End-user Simple Case](optd_web_page)
1.1. Optique

If the predefined queries do not cover the needs of the engineers, then IT experts are needed to provide the required flexibility of access. A lot of time can be lost to the back and forth communication between the IT experts and engineers, since the IT experts are not familiar with the domain of the data. Therefore accessing data in big companies can take several days. In data-intensive industries, engineers spend up to 80% of their time on data access problems. How much value could they create in that time?

Figure 1.2: End-user Complex Case

The Optique paradigm shift for data access, would apart form the direct cost of involving IT experts, also free up the time of the engineers. This would lead to even greater value creation. This paradigm shift is brought forth through the key feature that Optique utilizes, listed in the table below. Optique will bring about a paradigm shift for data access] These features will reduce the turnaround time for information requests, and absorb rather then through way, traditional rational database management systems (RDBM).

Table 1.1: Optique’s key features

1. Semantic end-to-end connection from the user to the data sources.
2. Intuitive queries using familiar vocabularies and conceptualizations.
3. Integration of multiple distributed data sources including streaming sources.
4. Massive parallelism salable far beyond traditional RDBMSs.
The Optique platform uses a meta ontology to capture user conceptualizations and declarative mappings, possible several if need be. This meta ontology is used by Optique to transform user queries into complete, correct and highly optimized queries over the data sources, in their original query constructs. With logic and reasoning, the Optique solution, through its transformations, enables engineers to do what otherwise had to be done with IT experts involved.

The platform will:
- Use an ontology and declarative mappings to capture user conceptualizations and to transform user queries into complete, correct and highly optimized queries over the data sources;
- Integrate distributed heterogeneous sources, including streams;
- Exploit massively parallel technologies and holistic optimizations to maximize performance;
- Include tools to support query formulation and ontology and mapping management;
- Use semi-automatic bootstrapping of ontologies and mappings, and query-driven ontology construction to minimize installation overhead.
1.1. Optique

Figure 1.5: Optique’s Architecture

Optique’s technical approach will exploit recent ground-breaking European research on semantic technologies, in particular related to query rewriting, combining this with techniques for scaling up query evaluation, in particular massive parallelism. These approaches will be integrated in a comprehensive and extensible platform that builds on open standards and protocols such as RDF, OWL, SPARQL, the OWL API, and the openRDF project.

The Optique platform will be tested and evaluated on two large-scale case studies from the energy sector:

"In the Siemens scenario, diagnosis engineers in their service centers for power plants try to detect events from time-stamped sensor data. To operate their tools for visualization and trend detection they need to query several TB of sensor data and several GB of data about events, such as “alarm triggered at time T,” distributed across several databases. With a 30 GB daily growth the total amount of raw data even exceeds what they can currently record."

"In the Statoil scenario, experts in geology and geophysics develop stratigraphic models of unexplored areas on the basis of data acquired from previous operations at nearby geographical locations. To feed data into their advanced visual analytics tool they need to query a pool of more than 1000 TB of relational data, structured according to several schemes with a total of more than 2,000 tables distributed across several databases."
1.2 OptiqueVQS

Optique has introduced the novel ontology-based visual query system OptiqueVQS, for the end-users. It takes part in Optique’s use of ontologies as a natural communication medium between end users and the databases. An intuitive data access tool like this, that directly engage domain experts with data, could substantially increase competitiveness and profitability in enterprise settings. [Soy+16, See Abstract]

OptiqueVQS has a user-centric design supported by a widget-based flexible and extensible architecture allowing multiple coordinated representation and interaction paradigms to be employed. The results of a usability experiment performed with non-expert users suggest that OptiqueVQS provides a decent level of expressivity and high usability, and hence is quite promising. [Soy+16, See Abstract]

Figure 1.6: Optique VQS Linear Shaped Query

"A VQS is not expected to be fully expressive; this is due to the fact that advanced query constructs, even in visual form, could be hard to comprehend and use for end users, while for IT experts textual mode would probably be more efficient and comfortable. In this respect, only domain and query constructs, which are frequently used and have a reasonable user perceived complexity, are realised. Perceived user complexity plays a binding role, since a visually expressed domain or query construct is virtually non-existent, even counterproductive, if end users are not able to comprehend and use it." [Soy+16, 5.2 Expressiveness, second paragraph]

The queries that can be made with OptiqueVQS is as of this writing, limited to linear and tree-shaped conjunctive queries. This means that the ranking method
can be exhaustive, since the complexity that it needs to deal with is limited to branching. Optique exploits the graph-based organization of ontological elements and data, for representing the domain and query structures, query by navigation. From a logic perspective, it uses ontological axioms to constrain the behavior of the interface, and to extend the available knowledge. [Soy+14, 2.2 Formal description]

Formally all the queries made with OptiqueVQS are subgraphs of a labeled directed RDF graph \( G = (N, E) \). This graph is made up of a finite set of labeled nodes \( N \) together with a finite set of labeled edges \( E \). The nodes represents objects and edges represents either object properties or data properties of the objects. All these resources are uniquely identified in their own pairwise disjoint alphabets, alphabets that are in the set of all the alphabets used \( U \). Either they are from our own alphabet of domain concepts, or logical concepts of the OWL 2 profile OWL 2 QL, the profile that OptiqueVQS supports.

An edge, in a subgraph/query made by OptiqueVQS, is a triple written on the form of \( \langle s, p, o \rangle \in (U \cup B) \times U \times (U \cup L \cup B) \). The "s" here is the subject, "p" is the predicate and "o" is the object of the triple. The meaning is that the subject can be a class or a blank, the predicate can only be a class, and the object can be a class, blank or literal. \( L \) is a set of uniform resource identifiers, resources like strings and integers, they are terminal literals. \( B \) is a set of blank nodes. [Soy+16, 6.1 Formal behavior]

SPARQL queries made by OptiqueVQS begins with PREFIX declarations denoting their respective namespaces, namespaces are for keeping identifiers from clashing. After that, comes the query form, the query form specifies the
1.2. OptiqueVQS

type of query, which is here SELECT. The output that the query will return is
modified by DISTINCT, for distinct entries. The output depend on the data
property constraints, the FILTERs that is put on the query. The WHERE
block contains the graph pattern that will be matched against the graph of the
model. Concepts without the right connections in the graph G will be rejected.
With the FILTERs some them with certain values will be kept.

The graph pattern in the WHERE block, made up of triple patterns, const-
ains the variables who’s bindings will be outputted. SPARQL has plenty
of features, but only those used by OptiqueVQS is of concern. OptiqueVQS
will only generate graph patterns made up of triple patterns on the form:
\( \langle x_i, a, m \rangle = V ar \times U \times (U \cup V ar \cup L) \). It generates a new variable \( x_i \) for
each new node added graphically. Any data constraint on some existing node
will only reuse an existing variable, not create new ones. Variables are blank
nodes, referable so that the constraints can be constrained further. [Soy+16
6.1 Formal behavior]

![Figure 1.8: Optique VQS Kernel and Pivot](Soy+16, See Fig. 3]

When starting out building, the end-user has an empty query, and as he or
she interact graphically, he or she is indirectly writing in SPARQL. After the
first step, which is selecting the kernel concept, any future actions will constrain
this kernel class further. Behind the scene, when selecting the kernel concept,
OptiqueVQS adds a triple pattern to the query, on the form: \( \langle x_1, rdf:type, v \rangle \in V ar \times U \times U \). The query’s first blank node \( x_1 \) have now been generated, and it
will match all instances of type \( v \), represented graphically by the root node.
The pivot is the graphically selected node, it is variable the OptiqueVQS shows possible extension for, which are the ones need sorted for better usability. The end-user has three ways of constraining the pivot. The first is to add a node to it though a labeled edge, this demands that pivot instances have a certain relation to some other instant of given type. This action results in two new triples on the form: \( \langle x_i, o, x_{i+1} \rangle \in \text{Var} \times U \times \text{Var} \) and \( \langle x_{i+1}, \text{rdf:type}, w \rangle \in \text{Var} \times U \times U \), being written to the query.
The second way to constrain the pivot is to add data properties to the class, demanding that instances have certain data values of some type, in certain ranges for example. Adding such constraints will add a new triple on the form \( \langle x_i, d, y \rangle \in Var \times U \times (Var \cup L) \) to the query. The third way of constraining the pivot is to select a subclass, which is an option suggested by default, it will generate a new triple on the form: \( \langle x_i, \text{rdf:type}, c \rangle \in Var \times U \times U \). [Soy+16, 6.1 Formal behavior]

1.3 Ontology-based Ranking

The problem that an ontology-based ranking method would solve for Optique VQS rather elegantly, is the problem of handling large ontologies. Any visual query system faces the problem of scalability against large ontologies. Optique VQS has to provide the end-user with the possible extension options, the elements of ontology: concepts and properties. The end-user only wants a certain extension at a time when iteratively constructing the query. [Soy+14, 1 Introduction, second last paragraph]

The problem for the end-user is that of irrelevant options burning the relevant ones, this start happening even for relatively small ontologies. For the number of concepts and properties to choose from increases drastically due to the wast amount of conclusion upon conclusion that ontological reasoning produces. This problem of choice hinders usability and therefor the value creation process of Optique. [Soy+14] 1 Introduction, second last paragraph]

The paper "Towards Exploiting Query History for Adaptive Ontology-based Visual Query Formulation" [Soy+14] suggests the adaptive solution of ontology-
Based ranking. This approach exploits a query history to rank and suggest ontology elements to a partially made query that the end-user has formulated so far. This means that the ranking method is context-aware, it takes into consideration the partial query to which the suggestions are for. [Soy+14, 1 Introduction, last paragraph]

Below is a part of the simplified Statoil ontology that will be used throughout this paper, from this chapter explaining the formal description of ontology-based ranking, too the experiment’s example query log and partial queries. The example query log used in the experiment was not generated with OptiqueVQS, so the queries contains some things that would not be theirs in a real log. There is no real example query log available to do the experiment with at the time of this writing, no real use-cases.

![Figure 1.11: Simplified Statoil Ontology.](Soy+14, See Fig. 2)

The ranking method takes a query log $QL$, which is a set of SPARQL queries: $QL = \{Q_1, Q_2, ..., Q_N\}$ constructed with OptiqueVQS. Then it calculates the conditional probabilities of each extension seen on queries, in the log, that matches the pattern of the partially made one. How this is implemented is presented in the chapter Ontology-based Ranker, not to be confused with this
1.3. Ontology-based Ranking

This chapter goes through the more formal definitions of are of concern. [Soy+14, 3.2 Ranking Method]

The probability of a partial query plus extension occurring within a query log, is the number of queries that the partial query plus extension is a part of, divided by the total number of queries. The query log is a set of text files written in SPARQL syntax, these files needs to be parsed and have their basic query pattern extracted. The basic patterns are the relevant part of the queries, the triples in the WHERE block. These triples need to be instantiated with the number of the query from which they came from.

All these triples make up the full graph of which the partial query can query for conditional matches. The triples from a query in the query log, which makes up its basic pattern, will contain variables. This basic pattern is turn into a graph when the variables are made into resources as they are instantiated. A function that takes a query log as input and returns its graph is therefor needed. Having converted the queries into graph patterns, the instantiation can take place.
1.3. Ontology-based Ranking

The probability of a graph pattern plus extension occurring within the log, is the number of graph patterns in the log that are supergraphs of the graph pattern plus extension, divided by the total number of graph patterns in the log. After converting the query files into basic graph patterns, the ranking method needs a way to determine if a graph pattern is the supergraphs of a partial query plus extension. The available extensions are basic patterns themselves, either constituted of two triple patterns for a branch, or one triple for a data property, connected to the rest of the partial query’s basic graph pattern.

As the end-user constructs his or hers query, for each new addition from the kernel to the last pivot, the ranker must at each step find the extensions already seen on the given partial query, and calculate their conditional probability. These extensions then have their rank, the probability is used as rank. The extensions that are not seen on the given partial query, in the log, but are possible for the end-user to extend the query with, will have the rank of zero probability.

Improved versions of the basic ontology-based ranking method, would not only find the exact matches of the partial query. It would also let extensions,
found on similar partial query matches, weigh in. How much extensions on these similar queries would weigh in, depends on the similarity metric used. There are different similarity dimensions that queries can be compared with. There is at least four dimensions, not concerned with semantics; structure-based, content-based, language-based, and result-based similarity. [DG13 pp. 5.1, 5.2, 5.3, 5.4] Semantic-distance is proposed and seen in this chapters example. [Soy+14 5 Discussion, Semantics distance:] 

For a formal description of what the ranking method calculates, see the following two equations. $QL$ is the query log, $\cdots$ is the cardinality, $\{ \cdots | \cdots \}$ describes a set, it uses $|$ to say "such that". $p(Q_i)$ is all sub-parts of query $i$ of the query log, $P$ is the partial query, and $T$ is the extension who’s conditional probability is being calculated.

$$Pr(P) = \frac{|\{Q_i \in QL | P \subseteq p(Q_i)\}|}{|QL|} \tag{1.1}$$

[Soy+16 See Eq. 1]

**Explanation** The probability $Pr$ of partial query $P$ equals the number of elements in the set of all queries in the query log, that has the partial query as a part of itself, divided by the total number of queries in the query log.

$$Pr(T|P) = \frac{Pr(T \cup P)}{Pr(P)} \tag{1.2}$$

[Soy+16 See Eq. 2]

**Explanation** The probability $Pr$ of an extension $T$ given the partial query $P$, equals the probability of the partial query plus extension divided by the probability of the partial query alone.

A graph pattern from a partial query, is a subgraph of a query in the log, if all its triple patterns are covered by that of the log query’s set of triples. A triple pattern from a query from the query log, covers a triple pattern from the partial query, if all the connections are covered. It does not matter what the blank nodes are called, only that there exists an instance in the other set with the same connections. To make sure that we get the correct supergraphs and their extensions, the names of the variables, of the different queries, must be concatenated with an identifier of the query it came from. A modified partial query can then query the triple store of instantiated triples.
Table 1.2: Ontology-based Ranking Steps

1. Parse the queries in the query log.
2. Instantiate the triples of each query.
3. Add the instantiated triples into a triple store.
4. Parse the partial query from the end-user.
5. Modify it to extract the two types of extensions.
6. Accumulate instances of the same extension.
7. Divide the matching queries with the total.
8. Divide the extension with the total of extensions.
9. Divide these two to get the conditional probability.
10. Do this for every unique extension found.
11. List of extensions and their ranks/probability.

[Soy+14, See Ranking Method]

The partial query needs to be modified to make it extract seen extensions. This modification is done by adding a triple pattern on the form ⟨?pivot, ?property, ?variable⟩. Where ?pivot is the variable of the pivot, ?property is either an object or data property to some variable ?variable found in the matching supergraphs. An OPTIONAL needs also be added, it has the triple ⟨?variable, rdf:type, ?type⟩. The query will now optionally output what class ?type that the object properties are constraining the variables too.

The output form of the partial query is also modified. The modified partial query outputs the instances being constrained, the property of the constrain, and optionally the class of the instance being constrained too. Since the instantiated variables in the model of the query log, have modified names, names made up of query number plus variable identifier, on can do SELECT DISTINCT and start calculating the probabilities. The DISTINCT is also a modification that this approach does to the partial query.

With the formal semantics that the Optique environment offer, a natural content based similarity measure is possible. Such a metric can be used to let similar queries increase each others probability, by for example letting extensions to subclasses count as extensions to their superclass. One can reason about semantic similarities like subclasses, sub-properties, inverses, and role chaining, to increase the common sense of the weights to be given. Semantic distance, when it comes to inheritance, is worth considering. With the basic implementation only exact matches contribute too the probabilities.
1.3. Ontology-based Ranking

The last step of the ranking is to sort and divide up the ranked extensions. OptiqueVQS needs two lists of ranked suggestions. The first list is of object property suggestions, these suggestions are concept-relationship pairs, for the left weight. The second list is of data property suggestions, these suggestions are for the widget to the right. These lists are divided up into pages of a certain number, the number of entries that respective widgets displays. An entry being either a concept-relationship or data property, plus their rank/probability.

### Matches

| ?c3     | prop                  | ?type               | Pr(T|P) | Widget |
|---------|-----------------------|---------------------|-------|--------|
| _:q1c2  | T1                    | ns2:drillingFacility| 0.16  | W2     |
| _:q1c3  | T2                    | ns2:drillingOpCompany| 0.33  | W2     |
| _:q1a1  | T3                    | ns2:name            | 0.16  | W3     |
| _:q3c2  | T4                    | ns2:drillingOpCompany| 0.16  | W2     |
| _:q3c3  | T5                    | ns2:drillingFacility| 0.16  | W2     |
| _:q3a1  | T5                    | ns2:wellboreContent |       | W3     |

**Figure 1.14: Modified Partial Query and Possible Extensions**

[Soy+14, See Fig. 5]
The Resource Description Framework (RDF) is a framework for representing information. It is a standard made by the World Wide Web Consortium (W3C), an international community that work together to develop Web standards. These standards, the W3C standards, define an Open Web Platform for web application development. RDF is part of this platform, a part that finds itself in what is called the Semantic Web stack, an addition to the classic ‘Web of documents’. This stack of technologies came about because of W3C’s vision of the Web of linked data.

The technologies in the Semantic Web stack were made to make all the resources on the web have meaning. With semantics formalized in RDF, we get computer readable linked data of all the resources that are out there on the internet. The computers are able to reason their way to what is needed, and with all resources having unique URI’s (Uniform Resource identifier), which double down as URL’s, the computers knows where to find the resource.

The goal of the Semantic Web has not been reached fully yet. This vision is large in scope and involves having enough people actually formalizing their data correctly so that developers can make useful semantic web applications. For the vision to truly come true, ways have to be found, to make sure that untrustworthy linked data does not contaminate the rest, making the computers reach false or less useful conclusions.

Even if the Semantic Web is not fully realized yet, more closed applications can utilize the technologies from it, such as RDF, SPARQL and OWL. Closed applications meaning applications where the linked data used is from known
trustworthy sources, like Optique. Optique’s linked data is from the people that use the Optique platform themselves, so they only have to trust themselves, when it comes to the data.

2.1 RDF

The Resource Description Framework (RDF) is a framework for expressing information about resources. Resources can stand for anything, they are simply strings, commonly on URL form, because RDF is a web technology for dealing with resources located on the web. The meaning of the strings are stated with RDF triples, meaning here is just strings linked together through other strings. Some strings are reserved by RDF, RDFS and OWL, these are the strings that makes up the logic, that reasoners can generate conclusions with. [Conm, 1. Introduction]

Figure 2.2: RDF Logo

[Cont, 2. W3C Buttons (blue)]

URIs or IRIs (Internationalized Resource Identifiers) are interlinked with one another with triples. RDF allows for making statements about resources with these triples. The format of these statements are subject, then predicate, and object. Meaning that the subject has a relation to the object that is of the predicate. I, the subject, am, the predicate, writing, the object. Any concept can have its URI, and can be linked together with others as one wishes, but the reasoning afterwards is objective. [Conk, 1.1 Graph-based Data Model]

Below there are two triples with the same subject. These two triples are from the query instantiated with the number 2934. One can see that the query’s basic graph pattern connected a variable "well" to a variable "w" through the predicate 'hasWellbore'. The two variables have become instances themselves after being numbered. Furthermore one can see a third variable 'pos' being the object of "well"'s 'locatedIn'.

<_:q2934well> <http://purl.org/net/grafli#hasWellbore> <_:q2934w> ;
<http://purl.org/net/grafli#locatedIn> <_:q2934pos> .

An RDF graph, is a set of RDF triples. The subject can be either an IRI or a blank node, the predicate can only be an IRI. The reason for this is just that, logic with blank node predicates would be come very complex, and we can still model what we want without this feature. The object can be either an IRI, literal or a blank node. Literals are used for values such as strings, numbers, and dates. Blank nodes are local identifiers, always locally scoped to the file or RDF store holding the statements. [Conl, 3. RDF Graphs]
2.2 RDFS

RDF Schema provides a data-modelling vocabulary for RDF data. RDF Schema is a semantic extension of the basic RDF vocabulary. It provides mechanisms for describing classes of related resources and the relationships between these classes. RDF Schema describes properties in terms of the classes, if an instance falls into a class, then it has all properties associated with that class. The ’rdf:type’ property is used to state that a resource is an instance of a class. [Cono1. Introduction and 2. Classes]

Below there is a little sample of the Statoil example ontology labeled "Subsurface Exploration Ontology". Both RDFS and OWL is being used. Two mentionable RDFS concepts used here are "rdfs:range" and "rdfs:domain". These two are used to specify the range and domain of functions. For example the function "valueInOriginalUnit" can only point to decimals. With the use of domain and range, reasoners will conclude that "concededBy" is a function, since things with domains and ranges are functions. Another example of RDFS logic is sub-classes inheriting all the properties of their super classes.

@prefix : <http://www.optique-project.eu/ontology/subsurface-exploration/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

<http://www.optique-project.eu/ontology/subsurface-exploration> a owl:Ontology ;
  owl:versionIRI :2016-05-25 ;
  rdfs:label "Subsurface Exploration Ontology" .

# Annotation properties
# http://www.optique-project.eu/ontology
# /subsrfce-exploration/valueInOriginalUnit
:valueInOriginalUnit a owl:AnnotationProperty ;
  rdfs:range xsd:decimal .

# Object Properties
# http://www.optique-project.eu/ontology/
# /subsrfce-exploration/concededBy
:concededBy a owl:ObjectProperty , owl:FunctionalProperty ;
  rdfs:domain :Concession .

While RDF describes the concept of, for example, "rdf:property" as a relation between subject resources and object resources, RDFS extends this with the "rdfs:subPropertyOf" property. RDFS provides basic facilities like "rdfs:domain" and "rdfs:range", but do not provide a direct way to indicate property restrictions, properties that instances of a class cannot have. Direct support for such
declarations are provided by richer Web Ontology Languages such as OWL.

2.3 OWL 2

The OWL 2 Web Ontology Language is for formally defining meaning. OWL 2 ontologies provide classes, properties, individuals, and data values. The OBDA framework behind OptiqueVQS supports OWL 2 QL and a conjunctive fragment of SPARQL 1.1. OWL 2 QL is a profile of OWL 2 and in this profile query answering can be implemented by rewriting queries into a standard relational query language. [Soy+14 2.2 Formal description first paragraph]

The figure below gives an overview of the OWL 2 language. It shows OWL 2’s main building blocks and their relation to one another. Firstly some type of serialization is needed, from Manchester syntax to the Turtle format, only one of these is needed, Turtle is used here. Turtle documents are produced and parsed to serialize and exchange ontologies. OWL syntax is imported and parsed to construct the semantics which can be viewed as direct ontology structure or as an RDF-Based Graph. [Cong 2 Overview]
2.3. OWL 2

Figure 2.4: The Structure of OWL 2

"OWL 2 QL enables conjunctive queries to be answered in LogSpace (more precisely, \( AC^0 \)) using standard relational database technology; it is particularly suitable for applications where relatively lightweight ontologies are used to organize large numbers of individuals and where it is useful or necessary to access the data directly via relational queries (e.g., SQL)." [Coni, 2.4 Profiles]

Again form the Statoil example ontology "Subsurface Exploration Ontology". Some OWL concepts have been used, like "owl:Ontology" which simply is the concept of an OWL ontology. The resource "subsurface-exploration" is now an ontology. The resource "valueInOriginalUnit" is an annotation property. Annotation properties are things like "rdfs:label", "rdfs:comment", "rdfs:seeAlso", information intended more for humans than machines. Further "concededBy" is a functional property of probably "conceded", and "versionIRI" points to the resource standing for the version of the ontology. [Coni] [Quick Reference Guide]

```
<http://www.optique-project.eu/ontology/subsurface-exploration>
a owl:Ontology ;
owl:versionIRI :2016-05-25 ;
rdfs:label "Subsurface Exploration Ontology" .

# Annotation properties
# http://www.optique-project.eu/ontology
# /subsurface-exploration/valueInOriginalUnit

:valueInOriginalUnit a owl:AnnotationProperty ;
rdfs:range xsd:decimal .
```
2.4. TURTLE

# Object Properties
# http://www.optique-project.eu/ontology/
#     subsurface-exploration/concededBy

:concededBy a owl:ObjectProperty, owl:FunctionalProperty;
    rdfs:domain :Concession .

There are three basic notions in OWL 2 for modeling knowledge. The notion
of axioms, the basic statements that an OWL ontology expresses. Then there
is the notion of entities, elements used to refer to real-world objects, and lastly
the notion of expressions, combinations of entities to form complex descriptions
from basic ones. Then there are a bunch of advanced notions to do with classes,
properties, and individuals. [Conh] 3 Modeling Knowledge: Basic Notions

2.4 TURTLE

Turtle is RDF Triple Language that can be seen used throughout this paper.
This syntax allows for writing down RDF graphs in a compact textual format.
A TURTLE document like the sample here below, begins with namespace
pseudonyms for easier writing, one can use '@base' instead of the empty prefix:
'@prefix : <...>'. Turtle allows for multiple reuse of a subject with semicolon,
and reuse of a predicate with comma. [Conn] Terse RDF Triple Language

@base <http://www.optique-project.eu/ontology
    /subsurface-exploration/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
:concededBy a owl:ObjectProperty, owl:FunctionalProperty;
    rdfs:domain :Concession .

TURTLE uses the file extension '.ttl'. It is recommended by W3C that
Turtle files have this extensions, all lowercase, on all platforms. [Cond] File
Extension

2.5 SPARQL

The SPARQL 1.1 set of specifications that does not just provide a language
for writing queries on RDF data, but also protocols to query and manipulate
RDF graph content. This will be exploited in the implementation through
Apache Jena’s 'RDFConnectionFactory', to connect to a different HTTP service
endpoint that process SPARQL quires more efficiently. [Conq] 2 SPARQL 1.1
Query Language
2.5. SPARQL

It is recommended that SPARQL files have the extension ".rq" (all lowercase) on all platforms. The query log used ".q".

Figure 2.5: SPARQL Logo

As with TURTLE, the namespaces are given abbreviations, prefix bindings. Some standard prefix bindings are assumed unless otherwise stated, these are rdf, rdfs, xsd, fn, and sfn. SPARQL uses Turtle data format, allowing for IRIs to be abbreviated with prefixes for example. The result of a query is a solution sequence, corresponding to the ways in which the query’s graph pattern matches the data.

Each solution gives one way in which the selected variables can be bound to RDF terms so that the query pattern matches the data. The result set gives all possible solutions. Basic graph patterns are sets of triple patterns. Graph pattern matching is defined in terms of combining the results from matching basic graph patterns. A sequence of triple patterns, with the optional use of using filters, comprises a single basic graph pattern. Any other graph pattern terminates a basic graph pattern.

A FILTER is a restriction on the solutions over the whole group in which the filter appears. OptiqueVQS does not make complex queries, it keeps its queries to basic graph patterns with some filters that are not of interest when it comes to the ranking of possible extensions. The filters in basic graph pattern, constrain the one and only group that is the graph pattern when the query is just a basic graph pattern.

Below is the first query in the query log being used for the experiment later.

What happens with this query is that the instances that fit the pattern of the five variables, will be selected, and will make up the columns in the result set. The variables not selected will not be outputted. The FILTER will remove any pattern binding that does make its boolean expression true, which means in this case, that the latitude must be greater than 60, less than 64, and the longitude must be between 2 and 6.

PREFIX : <http://purl.org/net/grafli#>

```
SELECT ?wellbore ?unit ?column ?lat ?long WHERE {
  ?w a :Wellbore ;
  :hasWellboreInterval ?int;
  :name ?wellbore .
  ?int a :StratigraphicZone ;
  :hasUnit ?u .
```
2.6. Jena

The first variable "?wellbore" is the name of a "Wellbore" that has a wellbore interval that is a stratigraphic zone, that has a named unit of something that is of a named stratigraphic column. The wellbore that we are selecting the name of, is the wellbore of something that also has a "locatedIn". The thing that this something is located in, needs to have latitude and longitude. A wellbore is a hole that is drilled to aid in the exploration of new oil and gas pockets. Stratigraphy is the branch of geology that studies rock layers, the layers are refereed to as strata.

SPARQL has four query forms, these are SELECT, CONSTRUCT, ASK and DESCRIBE. Only SELECT is used in the query logs, but in setup of the experiment a CONSTRUCT query found itself of use. A SELECT query returns variables and their bindings directly. While the CONSTRUCT query returns an RDF graph specified by the graph template, that also has all the connections specified in the WHERE block. [Cona, 10 Query Forms]

Table 2.1: SPARQL Query Forms

<table>
<thead>
<tr>
<th>Query Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>Returns variables bound in matches of a query pattern.</td>
</tr>
<tr>
<td>CONSTRUCT</td>
<td>Returns the RDF graph of a set of triple templates.</td>
</tr>
<tr>
<td>ASK</td>
<td>Returns True if the query pattern matched, False otherwise.</td>
</tr>
<tr>
<td>DESCRIBE</td>
<td>Returns an RDF graph about matching resources.</td>
</tr>
</tbody>
</table>

2.6 Jena

Apache Jena is a free and open source Java framework for building semantic web and linked data applications. Its RDF application programming interface can be used to create and read RDF graphs, and serialize triples using a format like Turtle. ARQ is Jena’s SPARQL processor, with it, one can query ones RDF data. With its API it is possible to build queries programmatically. Apache Jena is the framework that was used in most of implementation in this paper, to do things like visit the log queries and the partial query in order to modify them. [Jend] Jena Home

Jena has many implementations of different concepts from the Semantic Web. One of core concept implemented in Jena is the concept of a model, which is represented by the class Model. The class "Model" is the primary container of RDF information contained in graph form, in Jena. One of Model’s other roles
2.6. Jena

is to provide an abstraction over different ways of storing the RDF nodes and relations. It has a rich Java API with many convenience methods. Under the hood, a Model instance can be in-memory data structures, disk-based persistent stores, with different inference engines. [Jena](#) The core RDF API

![Jena](#)

Below is a simple example of using "ModelFactory" to create a model, it has the convenient methods that can read and write RDF documents. If no document type is explicitly given in a second argument, it tries to read the suffix. Model understands the extension ".ttl" to be Turtle. "QueryFactory" is used to create queries, and "QueryExecution" is used to apply the query to a model. We can see that Jena’s is fond of factory design pattern. "ResultSetFormatter" is used to format "ResultSet" for output, in the case of a SELECT query it looks at its form and uses its projection variables as columns, in their respective order.

```java
import org.apache.jena.query.Query;
import org.apache.jena.query.ResultSet;
import org.apache.jena.rdf.model.Model;

Model model = ModelFactory.createDefaultModel();
model.read("data.ttl");

Query query = QueryFactory.create(queryString);
QueryExecution qexec =
    QueryExecutionFactory.create(query, model);
ResultSet result = qexec.execSelect();

ResultSetFormatter.out(System.out, result, query);
```

ARQ is a query engine for Jena. With ARQ’s API it is possible to manipulate queries on the syntax level or down to the SPARQL algebra. A SPARQL query in ARQ goes through several stages of processing. The query string is parsed, then the query is translated to a SPARQL algebra expression. This expression is optimized before ARQ devices a query plan and low-level optimization. Now the query plan can be evaluated. [ARQ](#) SPARQL Algebra
The application programming interface for ARQ is in the package "org.apache.jena.query". It contains the key classes for handling queries. If one wants to manipulate queries, packages like "org.apache.jena.graph" and "org.apache.jena.sparql" comes in handy. The class "Query" represents application queries, it is a container for all the details of the query. Results from a SELECT query is handled by a "ResultSet" instance, the class "QuerySolution" represents a single solution from such an instance. [Jen, ARQ API]

Fuseki is Apache Jena’s SAPIRQL server. It can be run as a Java Servlet on Apache Tomcat, just download the WAR file (Web Application Resource) and add it. The WAR file contains all the resource needed by Fuseki, including the JAR file to be executed. Fuseki support the SPARQL 1.1 protocols for query and update as well as the SPARQL Graph Store protocol. Its security is provided by Apache Shiro, and it has a user interface for server monitoring and administration. [Jen, Fuseki]

The three Jena components RDF API, ARQ/SPARQL API and Fuseki are the components of most direct concern when it comes to the implementation of the ranker. In the figure below, one can see where these components fit in, in the larger framework of Jena.
The Javadoc for Jena’s core RDF API: [Jenb, Javadoc Jena Core API] The Javadoc for the ARQ API: [Jena, Javadoc Jena ARQ API] For a tutorial on manipulating SPARQL queries with ARQ: [Jenab, Manipulating SPARQL Queries] All these are very useful, but make sure that you are looking at the newest version of the Javadoc, things get deprecated. Another little things to know; to disable the default logger under testing, which is Log4j, do this:

```java
org.apache.log4j.Logger.getRootLogger().
    .setLevel(org.apache.log4j.Level.OFF);
```

### 2.7 RDF4J

Eclipse RDF4J is another framework parallel with Apache Jena, both provided standard development kits for Semantic Web applications and their own SPARQL server, with triple store and support for the SPARQL protocols. [RDFb] About RDF4J] Under the making and testing of the basic ontology-based ranker, it became apparent that the queering for the extensions took way
to much time. With a model of all the instantiated triples in the example query log, some queries took to long of a time to wait for.

After some more testing, it was found out that it was Apache Jena that did not optimize the queries correctly, there was no fundamental reason for the queries to take the time that they did, with the model used. At that time the ranker only queried a local model, so to Fuseki was introduced and used through a RDFConnection, this too not only see if it handled the queries better, but also to make the ranker more general by separating the triple store away from the implementation. Apache’s own Fuseki didn’t handle the queries any better.

Eclipse RDF4J was introduced as a third party SPARQL server, all the Jena application logic was still good with only small modification. RDF4J’s SPARQL server running as an Java servlet on Tomcat, preformed with the efficiency that one would expect from querying a relatively small query log model, with queries with single tree structured basic graph patterns.

The WAR files for RDF4J’s server and workbench, are found included in the RDF4J SDK packages. Download RDF4J SDK] Download and decompress. In order to know where to point the RDF connection factory too, when making and RDFConnection, so that one can utilize all of the SPARQL protocols in communication with the given RDF database, the REST API of the server must be studied. The REST API for the RDF4J server can be studied here: REST API]

The SPARQL protocols and SPARQL graphs store protocol, are used in an RDF connection, but the endpoints too the different services must be known to the RDF connection factory. It might be enough to specify only the root URL to the server, as it is with Fuseki since it provides the standard Apache endpoints. But with the RDF4J server one needs give as arguments the query, the update and the graph store protocol endpoint URLs. RDFConnectionFactory]
This is how it looks like to connect to the RDF4J’s endpoints.

```java
RDFConnection conn = RDFConnectionFactory.connect(
    "http://localhost:8080/rdf4j-server/repositories/optique",
    "http://localhost:8080/rdf4j-server/repositories/optique"
    + "/statements",
    "http://localhost:8080/rdf4j-server/repositories/optique"
    + "/rdf-graphs/service);
```

For good measure, here is also how it looks like to connect to Fuseki’s endpoints.

```java
RDFConnection conn = RDFConnectionFactory.connect(
    "http://localhost:3030/optique/query",
    "http://localhost:3030/optique/data);
```

When an RDF connection is established it is possible for example load in a whole RDF document with all its triples. Not only that, one can for example also use the update service to make a DELETE query. In the example below, the DELETE query would simply delete all the triples in the triple store, but the repository as it is called with RDF4J, would not be deleted itself.

```java
conn.load("model.ttl");
```

Back to the inefficient query execution of Jena. The following is three versions of the same partial query, modified to fetch extensions seen on the same construct. The basic graph pattern in all three are the same, the only difference between the three are the order in which the triples appear. The first version of the query is the original, the triples appear as they appeared. The second version is sorted, meaning the triples closer too the root variable comes first, then in order the rest. The third version is shuffled in a way, to make the triples appear in an order as inconvenient as possible. The shuffle was done by trying to place the triples that are far away from the root more at the beginning, but also triples linked close together more away from each other in the query.

The original version of the query:

```sparql
PREFIX : <http://purl.org/net/grafli#>
SELECT DISTINCT ?cons ?prop ?type
WHERE
{
    ?bmd a :MeasuredDepth .
    ?c :hasCoreSample ?s .
    ?ttvd a :TrueVerticalDepth .
    ?wi :hasTopDepthMeasurement ?ttvd ;
         :hasBottomDepthMeasurement ?bmd .
    ?w a :Wellbore .
}
```

---

2.7. RDF4J
The sorted version of the query:

```
PREFIX : <http://purl.org/net/grafli#>

SELECT DISTINCT ?cons ?prop ?type
WHERE {
  ?w :hasWellboreInterval ?wi .
  ?w a :Wellbore .
  ?w :name ?wName .
  ?c :hasCoreSample ?s .
  ?s :name ?coreSample_name .
  ?p :valueInOriginalUnit ?permeability .
  ?z :hasUnit ?u .
  ?u :name ?unit_name .
  ?tmd a :MeasuredDepth .
  ?tmd :valueInStandardUnit ?top_md .
  ?ttvd :valueInStandardUnit ?top_tvd .
  ?wi :hasTopDepthMeasurement ?ttvd .
  ?ttvd a :TrueVerticalDepth .
  ?ttvd :valueInStandardUnit ?top_tvd .
  ?wi :hasBottomDepthMeasurement ?btvd .
  ?btvd a :TrueVerticalDepth .
  ?btvd :valueInStandardUnit ?bot_tvd .
  ?bot_tvd :valueInStandardUnit ?bot_md .
  OPTIONAL
  { ?cons a ?type }
  FILTER strstarts(str(?cons), "_:q")
}
```
The shuffled version of the query:

```sparql
PREFIX : <http://purl.org/net/grafli#>
SELECT DISTINCT ?cons ?prop ?type
WHERE {
  ?bmd a :MeasuredDepth .
  ?w :name ?wName .
  ?tmd :valueInStandardUnit ?top_md .
  ?c :hasCoreSample ?s .
  ?s :name ?core_sample_name .
  ?p :valueInOriginalUnit ?permeability .
  ?z :hasUnit ?u .
  ?u :name ?unit_name .
  ?w :hasWellboreInterval ?wi .
  ?w a :Wellbore .
  ?tmd a :MeasuredDepth .
  ?wi :hasBottomDepthMeasurement ?btvd .
  ?ttvd :valueInStandardUnit ?top_tvd .
  ?btvd :valueInStandardUnit ?bot_tvd .
  ?btvd a :TrueVerticalDepth .
  ?wi :hasBottomDepthMeasurement ?bmd .
  ?ttvd a :TrueVerticalDepth .
  OPTIONAL
  { ?cons a ?type }
  FILTER strstarts(str(?cons), "_:q")
}
```

The original query takes awhile to execute with the Jena framework, but the shuffled takes even longer. If you sort your queries in the same way that the sorted version has been, then query time is comparable to that of querying with the RDF4J framework. The RDF4J handles all the different versions with the same efficiency, the reason that the original version took so much longer with the RDF4J framework is because it ran first, running it for examples third, it only takes 9 ms.
Table 2.2: Query Times in Milliseconds

<table>
<thead>
<tr>
<th></th>
<th>With Jena</th>
<th>With RDF4J</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Version</td>
<td>73556 ms</td>
<td>101 ms</td>
</tr>
<tr>
<td>Sorted Version</td>
<td>5 ms</td>
<td>7 ms</td>
</tr>
<tr>
<td>Shuffled Version</td>
<td>334263 ms</td>
<td>6 ms</td>
</tr>
</tbody>
</table>

With the Jena framework it took the original versions of the query, approximately 1.2 minutes to finish querying. With the shuffled version it took approximately 5.6 minutes. On the other hand, with the RDF4J framework all queries took approximately the same amount of time, if you factor out first time run initialization. The sorted version of the query, with the Jena framework, was just as fast as with RDF4J. The model that this was tested on had 1236 triples. Below are the first couple of triples from its Turtle document:

```
_:q2906bot_tvd a <http://purl.org/net/grafli#TrueVerticalDepth> ;
   <http://purl.org/net/grafli#valueInStandardUnit>
   _:q2906top_depth_tvd .

_:q2920w a <http://purl.org/net/grafli#Wellbore> ;
   <http://purl.org/net/grafli#hasWellboreInterval>
   _:q2920wlb_int .

_:q2924c <http://purl.org/net/grafli#extractedFrom>
   _:q2924ci ;
   <http://purl.org/net/grafli#hasCoreSample>
   _:q2924sample .
```

The difference between the sorted and shuffled versions of the query, with the Jena framework, was that the shuffled version took over sixty thousand times longer than that of the sorted. Below is how one would use the RDF4J framework to query with. RDF4J distinguishes between three different query types: tuple queries, graph queries and boolean queries. The different types differ in what results they produce. SELECT queries are tuple queries. [Foue, Programming with RDF4J]

```java
Repository db = new SailRepository(new MemoryStore());
db.initialize();

try (RepositoryConnection conn = db.getConnection()) {
    try (InputStream input =
         new FileInputStream(new File(modelString))) {
        conn.add(input, "\", RDFFormat.TURTLE );
    } catch (Exception e) {}{}

    TupleQuery query = conn.prepareTupleQuery(queryString);
    try (TupleQueryResult result = query.evaluate()) {
        while (result.hasNext()) {
            result.next();
        }
    }
}
```
To sum up, RDF4J was introduced because Jena does not optimize the queries correctly. RDF4J’s way of representing SELECT queries as tuple queries does not suffer this problem. Query execution in RDF4J and Jena is lazy, meaning that the execution does not happen all at once but only as the next solution is asked for. In the testing of both frameworks, it was needed to iterate over all the solution.

An educated guess on what Jena does wrong with its query execution, which makes shuffled queries so slow: Imagine that what it does, is to just take each triple in the query one by one, trying to match the basic pattern so far seen, as it appeared in the query. Beginning again if the pattern does not match anything any more. When the query is shuffled in the given way, doing so does not exclude many matches at the beginning, and with this strategy Jena is taking the product when trying matches. The complexity of taking the product in big O notation:

\[ O(n^2) \]  

2.8 Mahout

Apache Mahout provides an environment for creating machine learning applications. It has three major features. First feature is a programming environment and framework for building scalable algorithms. Second feature is the wide variety of premade algorithms for Scala with Apache Spark, H2O, Apache Flink. The third feature is Samsara, a vector math experimentation environment with R-like syntax which works at scale. [Mahé, What is Apache Mahout?]

![Figure 2.9: Apache Mahout Logo](MahH)

The collaborative filtering ranker does not need all of the fancy features of Apache Mahout, at least not for this paper’s purposes. When starting out with Mahout one might not need to go right for a distributed Hadoop-based recommendeder, its complexity is not necessary here. A non-disturbed
2.9 Protégé

Protégé is a free, open-source ontology editor and framework for building intelligent systems. With protégé one can construct domain models and knowledge-based applications with ontologies. Protégé Desktop allows one to create, explore and edit ontologies. It has lots of features, like reasoning support. Reasoners are plugins, protégé has lots of different types of plugins. The Pellet and FaCT++ reasoners have very good support, with direct interfaces.

Protégé desktop was used to explore the Statoil example ontology. For the purposes of studying its metrics, to see if the query log and ontology used here could benefit the ontology-based ranker over the collaborative filtering based ranker, or vice versa. For example, how does the relationship between the number of concepts and the number of connections, affect the performance of the different rankers. To be able to say something about this one have to have at least two ontologies to compare. A speculation would be that at least the collaborative filter would like the use of many concepts in each query, so to have a lot of collaboration, overlap.

The following are some of protégé’s views, looking at the Statoil ontology. First there is the metrics view of the ontology, listing how many occurrences there are of different things, the namespaces and the ontology IRI, version and annotations. Second is the classes, with the hierarchy, description and usage. Third is the object properties, in other words, the functions between classes, based on subPropertyOf assertions. Lastly is the data properties, functions from class to datatype, also based on subPropertyOf assertions.
2.9. Protégé

Figure 2.11: Statoil Ontology Metrics

Figure 2.12: Statoil Ontology Classes
Considered to implement more than the basic version of the ontology-based ranker, that would utilize reasoning, but with the particular query log and example ontology available, the inferred model was not particularly interesting. Below are the few relevant triples that were inferred with the Statoil schema available and with the same model used above. They are all typesetting conceptualized variables, which are used in the query instantiated with the number 2962. The rest of the triples are added for OWL and RDFS purposes, making up the logical framework that triggers inference. Only the first triple below is of actual interest, since it typesets a query variable concept to StratigraphicZone from the Statoil ontology.
To model with protégé one must be familiar with Manchester syntax, it is the syntax that protégé uses. Protégé is a nice tool for ontology development. Developing an ontology for a given domain formally defines a common vocabulary in which you can share domain information with others. Ontologies are great for making assumptions explicit, and having a way to assure a common semantic understanding. It also separates domain knowledge from operational knowledge, which then can be analyzed, and of course one can reason one's way to more knowledge. [FM01] Ontology Development 101
The figure above shows the class hierarchy of the Statoil ontology. This is an example of the many features of protégé provided by one of its plugins, in this case OWLViz. Another example of what can be done in Protégé, is for instance, executing SPARQL queries on the models loaded. If some feature is not provided by the default plugins, it might be covered by one of the plugins offered by the protégé plugin library. [proc, Protégé Plugin Library]

2.10 Others

**Eclipse**  The ontology-based ranker and collaborative filtering based ranker were developed on the Eclipse IDE. This integrated development environment is very extendable with plugins, from its marketplace and repositories. [Foug, Eclipse IDE]

![Eclipse Logo](Fouh, Eclipse Logo)

Figure 2.16: Eclipse Logo

**Maven**  Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project’s build, reporting and documentation from a central piece of information. M2Eclipse is the official Eclipse project for Maven integration for the Eclipse IDE. [Foua, Apache Maven]

![Maven Logo](Foub, Maven Logo)

Figure 2.17: Apache Maven Logo

**Tomcat**  The Apache Tomcat software is an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. [Fouc, Apache Tomcat]

![Tomcat Logo](Foud, Tomcat Logo)

Figure 2.18: Apache Tomcat Logo
PART II

Ranking Methods
CHAPTER 3

Ontology-based Ranker

This chapter goes through the interesting parts of the implementation of the basic ontology-based ranking method. The different sections below take a look at the logically separated components, classes that are being utilized by the ranker. These classes are QueryLog, LogModel, QueryVisitor, PartialQuery and QueryModel. Other than these main components, there was the RankingUtils class for miscellaneous methods, and then there was the interface Ranking.

Disclaimer, the code samples presented here are not the whole picture, and is more for illustrative purposes, showing examples of things to consider when implementing this type ranker, in the Jena framework. For example some strings have been inlined were the variables holding them otherwise would have been, this is to show clearer what is going on. Not to be mistaken for how it is in the actual code. Here is an example of some methods that the Ranking interface can have:

```java
public interface Ranking {
    public Set<Entry<List<Resource>, Double>> entries();
    public <K, V extends Comparable<? super V>> Map<K, V>
            sortbyValue(Map<K, V> map);
    public LinkedHashMap<List<Resource>, Double>
            asLinkedHashMap();
    public void makeRanking(QueryLog queryLog,
                              PartialQuery query);
    public int size();
}
```

The interface Ranking provides a common interface for both this ontology-based ranker and the collaborative filtering based ranker described in the next chapter. The experiment only deals with the Ranking interface, and does therefore not know which one of the two rankers it is dealing with. This interface is both for better design and for making sure that the experiment is under the same condition for both ranking methods.

```java
public class OntologyBasedRanking
        extends LinkedHashMap<List<Resource>, Double>
        implements Ranking
```
The class that puts it all to use, is the OntologyBasedRanking class. It inherits from the class LinkedHashMap, with the parameters; list of extensions, of both types that OptiqueVQS deals with, and extension probability. The implementation ended up relying on maps for dealing with the extensions, convenient for looking up occurrences. Linked hash map keeps an internal list of the extensions as they were inserted. One can sort the linked hash map on the probabilities, the values, or the key for that matter. Java takes care of the hash function for class List. [oracle Class LinkedHashMap<K,V>]

```java
public <K, V extends Comparable<? super V>> Map<K, V> sortByValue(Map<K, V> map) {
    return map.entrySet()
        .stream()
        .sorted(Map.Entry.comparingByValue(Collections.reverseOrder()))
        .collect(Collectors.toMap(
            Map.Entry::getKey,
            Map.Entry::getValue,
            (e1, e2) -> e1,
            LinkedHashMap::new
        ));
}
```

Relaying on directly inhering from classes is not good design pattern, as have been done here, do not try this at home. Rather, good programmers write in terms of interfaces not super classes, known as interface-based programming. This approach has been known for some time to produce more solid code, an exception where inheritance works on scale is widget hierarchies. When doing interface-based programming, you let your classes implement interfaces and keep an internal data field for the super classes that you would otherwise extend. [Hol Why extends is evil]

In the middle of implementing the ontology-based ranker, the ModelCom class that LogModel and QueryModel inherits from, broke the ranker after a new Jena update came out. One might think that this was deserved, as the implementation does not make use of the interface-based programming, if it had I could have more easily changed out the internal data field with another model. But then a new patch came along and fixed the problem, and no lesson were learned. Just extending and getting much for free can be very tempting.

Summarizing the sequence of stages that the ontology-based ranker goes through to produce the example ranking below, and the roles that the different components play. First the QueryLog class looks though the directory hierarchy given and reads in the queries found, making itself a map of query paths and the Query found at each one. Then this LogQuery is given over to LogModel, it makes QueryModels out of LogQuery’s queries, then adds them to its union. The partially constructed query from OptiqueVQS would then be given to the constructor of PartialQuery, modifying it with the help of QueryVistor to
extract extensions, a pivot must be given, or the first projection variable of the
SELECT is just used. [Jens, getProjectVars()]

PREFIX : <http://purl.org/net/grafli#>

SELECT DISTINCT ?cons ?prop ?type
WHERE
{
?w a :Wellbore .
?well :locatedIn ?pos .
?well :hasWellbore ?w .
?w ?prop ?cons
OPTIONAL
{
?cons a ?type
FILTER strstarts(str(?cons), "_:q")
}
}

Above is a partial query that has been modified to extract possible extensions.
The pivot that was used here, simulating the node that the end-user is con-
sidering constraining with an extension, is the variable "?w". The basic graph
patterns of the queries in the query log was instantiated and made into a model.
This model was then queried with the query above, which resulted in a total of
84 matches. Below is the list of all the unique types of extensions in the result
set, and their count (e.g. how many instances there were of each one).

Total number of matches: 84

name 27
hasWellboreInterval 25
hasWellboreInterval :StratigraphicZone 13
hasFormationPressure 6
hasWellboreInterval :FluidZone 5
hasFormationPressure :FormationPressure 3
hasMeasurement :PressureMeasurement 1
hasTotalLength 1
hasMeasurement :TemperatureMeasurement 1
hasMeasurement 1
wellboreDocument 1

When LogModel is queried, a set of matches is produced, these matches come
from the different queries. Some matches are the same suggested extension, just
from different queries. The insistence of the same extension are counted up.
Each type of extension has its count of occurrences in the result set. For each
type of extension, one divides the count with the total number of matches. The
total number of matches is the number of entries in the result set, each instance
counts. Below is the ranking that was calculated with the numbers above.

name 0.3214285714
hasWellboreInterval 0.2976190476

43
For the basic ontology-based ranker to perform well, it needs a large diverse query log. This is needed for the ontology-based ranker to be competitive, because it only matches exact similar basic patterns. It does not use a similarity measure to let other similar queries weigh in. In cases where the query log is small, or just does not contain an instance of the type that the end-user is constructing, simply no matches will be found, resulting in the basic ontology-based ranker being clueless of what to suggest.

### 3.1 Modification

Below there is a query from the query log, that was used by the experiment later, playing the part as a partial user query. It was given to one of the constructors of the "PartialQuery" class, and turned into an instance of it. This was done with the help of a "QueryVisitor" instance. It extracts the basic pattern from the original query to be used in the partial query for fetching extensions. The FILTERs are skipped over. The basic pattern is extracted as is and represented as an ElementPathBlock, that will be used in the partial query.

```
PREFIX : <http://purl.org/net/grafli#>

WHERE
{ ?w a :Wellbore ;
  :name ?wellbore ;
  :hasWellboreInterval ?int .
?c a :Core ;
  :extractedFrom ?int .
?int :hasTopDepthMeasurement ?top .
?top a :TrueVerticalDepth ;
  :valueInStandardUnit ?top_tvd_m .
?c :hasCoreSample ?s .
?c :hasCoreSample ?s2 .
?s2 :hasPorosityMeasurement ?por .
?por :valueInStandardUnit ?porosity .
?w :hasFormationPressure ?fp .
?fp :valueInStandardUnit ?formation_pressure ;
```
3.1. Modification

The query had the projection variables "?wellbore", "?c", "?top_tvd_m", "permeability", "?lat", and "?long". None of these are of use, that is if non of them are going to be used as the pivot, which will happen if no pivot is given. In a real setting, OptiqueVQS would provide the ranker with the pivot, that the end-user has selected. The pivot variable being used in this case is "?wellbore". The partial query gets added too it the triple <?wellbore, ?prop, ?cons>, in order to select ?prop, and the triple <?cons, a, ?type> is added in an optional, to optionally select the type of the object that is constraining subject.

PREFIX : <http://purl.org/net/grafli#>

SELECT DISTINCT ?cons ?prop ?type
WHERE
{ ?w a :Wellbore ;
 :name ?wellbore ;
 :hasWellboreInterval ?int .
 ?c a :Core ;
 :extractedFrom ?int .
 ?int :hasTopDepthMeasurement ?top .
 ?top a :TrueVerticalDepth ;
 :valueInStandardUnit ?top_tvd_m .
 ?c :hasCoreSample ?s .
 ?c :hasCoreSample ?s2 .
 ?s2 :hasPorosityMeasurement ?por .
 ?por :valueInStandardUnit ?porosity .
 ?w :hasFormationPressure ?fp .
 ?fp :valueInStandardUnit ?formation_pressure ;
 :hasDepthMeasurement ?fp_depth .
 ?int :hasTopDepthMeasurement ?top_md ;
 FILTER ( ( ?form_press_depth > ?top_md_m ) &&
 ( ?form_press_depth < ?bot_md_m ) )
 ?well :hasWellbore ?w ;
 :locatedIn ?pos .
 ?pos :latitude ?lat ;
 :longitude ?long
 FILTER ( ( ( ( ?lat > 60 ) && ( ?lat < 61 ) ) &&
 ( ?long > 2 ) ) && ( ?long < 3 ) )
}
3.1. Modification

The model of the query above has been turned into a model. Its variables have been instantiated with "_:q0" prefix plus their name. The underscore works as the namespace for these instantiated variables.

```sql
q0p valueInStandardUnit q0permeability
q0fp_depth valueInStandardUnit q0form_press_depth
q0top_md valueInStandardUnit q0top_md_m
q0s2 hasPorosityMeasurement q0por
q0s hasPermeabilityMeasurement q0p
q0w name q0wellbore
q0w hasWellboreInterval q0int
q0c extractedFrom q0int
q0bot_md valueInStandardUnit q0bot_md_m
q0top type TrueVerticalDepth
q0well hasWellbore q0w
q0fp hasDepthMeasurement q0fp_depth
q0top_md type MeasuredDepth
q0por valueInStandardUnit q0porosity
q0w type Wellbore
q0W hasFormationPressure q0fp
q0well locatedIn q0pos
q0c hasCoreSample q0s2
q0c type Core
q0top valueInStandardUnit q0top_tvd_m
q0int hasTopDepthMeasurement q0top
q0fp_depth type MeasuredDepth
q0bot_md type MeasuredDepth
q0int hasTopDepthMeasurement q0top_md
q0c hasCoreSample q0s
q0fp valueInStandardUnit q0formation_pressure
q0pos latitude q0lat
q0pos longitude q0long
```
3.1. Modification

q0int hasBottomDepthMeasurement q0bot_md

Some possible weaknesses with the approach suggested in the paper "Towards Exploiting Query History for Adaptive Ontology-Based Visual Query Formulation". [Soy+14, The Paper] Is that DISTINCT is not really needed, because even if a query in the query log had stated a triple twice it would not be added twice in any model of it, because models are sets, in sets things only exists or not. Furthermore, in the case where a query has for one variable, the same property too different variables of the same type, should that not count as two?

@c11 ns3:extractedFrom ?c12 .
@c12 a ns1:theMoon .
@c11 ns3:extractedFrom ?c99 .
@c99 a ns1:theMoon .

A further possible weakness is that extensions that are already on the pivot, of the query that the end-user is constructed, is suggested. To illustrated this, look at the following two queries. The first one, queries after things that are of type object, of the athenum namespace. These things has already been constrained to have the property ':will' to something. Now the second query has another extension that would be interesting for the end-user, but also the same extension already on the pivot.

This is the first query, it was used as the end-user’s partial query:

```sparql
PREFIX : <http://atheneum.org/schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?thing WHERE {
  ?thing a :object .
  ?thing :will ?beta .
}
```

This is the second query, it was the single query in the query log:

```sparql
PREFIX : <http://atheneum.org/schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?thing WHERE {
  ?thing a :object .
  ?thing :can ?alpha .
  ?thing :will ?beta .
}
3.1. Modification

Let the first query now be turned into a modified partial query, and the second query be turned into a QueryModel. A instance of QueryModel can be used to make a LogModel or just queried directly since it inherits model. The modified partial query looks now like this:

```sparql
PREFIX : <http://atheneum.org/schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?cons ?prop ?type
WHERE
  { ?thing rdf:type :object ;
    :will ?beta ;
    ?prop ?cons
    OPTIONAL
    { ?cons rdf:type ?type}
    FILTER strstarts(str(?cons), "_:q")}
```

The query log model is now about to be queried, with the modified partial query, consists of only the model of the second query. This log model consists of only three triples, and looks like this if printed out:

```
q0thing type object
q0thing will q0beta
q0thing can q0alpha
```

The result from applying the partial modified query to this single query query log, is that the query will select the "will" property extension, disregarding that it already is constraining the end-users partial query pivot. Using Jena’s ResultSetFormatter, the query result set looks like this:

```
| cons   | prop | type |
|----------------------------------|
| <_:q0beta> | :will |       |
| <_:q0alpha> | :can  |       |
```

The fact that extensions already on the pivot is selected, might not be that much of a problem since OptiqueVQS can just use the ones that it is not already seeing being used to constrain the pivot. Otherwise this potential problem could be solved both by adding FILTERs to the modified query itself or just let them be removed by the ranker before sending the ranking to OptiqueVQS.

The last possible weakness that should be looked at, and has been dealt by the addition of the special FILTER seen below. Is that of "rdf:type" is being suggested as an extension property, when this is of no use to OptiqueVQS. The FILTER solving this problem looks like this:

```
FILTER strstarts(str(?cons), "_:q")
```
Without this FILTER, result sets would contain extensions of the "rdf:type" property on the actual properties of the domain. The result sets would contain extension suggestion like the ones below, having been found bounded to "?cons" and "?prop":

<table>
<thead>
<tr>
<th>cons</th>
<th>prop</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Core</td>
<td>&lt;...w3.org...ns#type&gt;</td>
<td></td>
</tr>
<tr>
<td>:Wellbore</td>
<td>&lt;...w3.org...ns#type&gt;</td>
<td></td>
</tr>
<tr>
<td>:TrueVerticalDepth</td>
<td>&lt;...w3.org...ns#type&gt;</td>
<td></td>
</tr>
<tr>
<td>:MeasuredDepth</td>
<td>&lt;...w3.org...ns#type&gt;</td>
<td></td>
</tr>
</tbody>
</table>

The FILTER makes the SELECT query having only to keep the bindings where the variable "?cons" has been bounded to something that starts with "_:q". Having this FILTER assures that the result sets will only contain possible extensions of interest, like these type of extensions:

<table>
<thead>
<tr>
<th>cons</th>
<th>prop</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>_:q7985fp</td>
<td>:hasFormationPressure</td>
<td></td>
</tr>
<tr>
<td>_:q7985wi</td>
<td>:hasWellboreInterval</td>
<td>:StratigraphicZone</td>
</tr>
<tr>
<td>_:q6709s2</td>
<td>:hasCoreSample</td>
<td></td>
</tr>
<tr>
<td>_:q6771wi</td>
<td>:extractedFrom</td>
<td></td>
</tr>
</tbody>
</table>

A possible complication that could come about from the use of this FILTER, is that the triple store that the triples are loaded into, might change the naming of the resources. In the figure below ten triples have been queried out from Fuseki's triple store. A model with the naming convention "_:q" was loaded into Fuseki, but when looking at the triples when in the triple store, they have been changed to use "_:b". Using the FILTER results in all the possible extensions be filter out, since they do not start with "_:q" but "_:b", to get results the filter had to be commented out.
3.2 QueryLog extends LinkedHashMap<Path, Query>

That said, when actually using Fuseki or RDF4J with RDFConnection there were no problem with using the filter. The renaming above was experienced when using Fuseki’s server interface directly, to explore what different queries gave as result graphically. Now let’s look at some of the implementation details of the ontology-based ranker.

The class QueryLog is for representing the queries making up the query log, and for keeping track of where they were found and what they where called. It extends LinkedHashMap<Path, Query> to do this. A linked hash map has a predictable iteration order, it differ from HashMap in that it maintains a double linked list of all the entries. This list defines the iteration order, which is normally the order of in which the keys were inserted, but it can sorted on either key or value. [orac, Class LinkedHashMap<K,V>]

Path is an interface representing an object used to locate a file in the system dependent file system. A Path is hierarchical and is composed of the directories from the root component to a file or last directory, separated by a special delimiter. Accessing a file using an empty path is equivalent to accessing the current directory of the file system, from which the application process is running from. [orag, Interface Path]

Query is a class from the Jena framework, it is a data structure for a query as presented externally. Queries are parsed and manipulated through ARQ’s API, at the syntax level or the algebra level. You get very far by just keeping to the syntax level. The implementation in the beginning dabbled at the algebra level, but I quickly found out that the syntax level was enough. The 'QueryFactory.create(queryString)' parses query strings and creates the query

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>_b0</td>
<td>_b1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>_b2</td>
<td>_b3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>_b2</td>
<td>_b4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>_b5</td>
<td>_b6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>_b5</td>
<td>_b7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>_b8</td>
<td>_b9</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>_b8</td>
<td>_b10</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>_b11</td>
<td>:Wellbore</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>_b11</td>
<td>_b12</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>_b13</td>
<td>:MeasuredDepth</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 3.1: Triples loaded into Fuseki
3.2. QueryLog extends LinkedHashMap\langle Path, Query\rangle
data structure. Once a query is build it can be passed to QueryExecutionFactory
to produce an execution engine. [Jens, Class Query]  
The QueryLog needs the functionality of walking around in the file system, for
exploring the log directory tree for any SPARQL files. For this purpose QueryLog
has an inner class LogVisitor that extends the SimpleFileVisitor\langle Path\rangle. The
simple file visitor class is a simple visitor of files with default behavior to visit
all files and to re-throw I/O errors. One can override its methods like the
'visitFile' method, in this method it creates queries out of the SPARQL files
that it finds. [orae, Class SimpleFileVisitor\langle T\rangle]  
```java

class LogVisitor extends SimpleFileVisitor<Path> {
    PathMatcher matcher = FileSystems.getDefault().getPathMatcher("glob:**.q");

    @Override
    public FileVisitResult visitFile(Path path,
            BasicFileAttributes attr) {
        if (matcher.matches(path)) {
            try {
                String queryString = FileUtils.readFileToString(
                        new File(path.toUri()), StandardCharsets.UTF_8);
                put(path, QueryFactory.create(queryString));
            }
            catch (IOException e) {}
        }
        return FileVisitResult.CONTINUE;
    }

    PathMatcher matcher = FileSystems.getDefault().getPathMatcher("glob:**.q");
    Path startingPoint = FileSystems.getDefault().getPath(queryLogdir);

    try {
        Files.walkFileTree(startingPoint, new LogVisitor());
    } catch (IOException ioe) {
    }
}
```

The two asterisks in the regular expression "glob:**.q" matches all the
directories that one must go through to get to the query file with the extension
'.q'. The PathMatcher interface [orah, Interface PathMatcher] usually is used
to facilitate the default file matcher of the file system that the Java virtual
machine is running on. To get the PathMatcher one uses the FileSystems class,
which contains factory methods for file systems. [orab, Class FileSystems] The
FileUtils class from Apache commons has general file manipulation utilities,
like reading, writing, copying, making and deleting. [Combl, Class FileUtils]
3.3 LogModel extends ModelCom

To use a simple file visitor like QueryLog, one can use the static method from the class Files named 'walkFileTree'. The class Files consists exclusively of static methods that operate on files, directories, or other types of files. The method 'walkFileTree' as the name suggest walk though the given directory, using a file visitor from the starting point. It does not follow symbolic links. It will visit all levels of the file tree. [ora] Class Files

3.3 LogModel extends ModelCom

The class LogModel represents the full model of the query log, consisting of the union of all the QueryModels. QueryModel instantiates the query for LogModel. This class inherits from the class ModelCom. The ModelCom class is model implementation with just the most common methods implemented. Mainly these methods are convenience methods for model implementations, like LogModel. It is intended to be used as a base class, for sub classes to derive from. Other model classes include, InfModelImpl, MonitorModel and OntModelImpl. [Jen] Class ModelCom

The ModelCom base class constructor requires a graph, for basing the model on. Graph is an interface for an implementation maintaining collections of RDF triples. The core interface is small, with just methods like add, delete, find, contains, but is augmented by additional classes to handle more complicated matters. One simple example of a graph class is the collection graph. [Jen] Interface Graph

```
public LogModel() {
    super(new CollectionGraph());
}
```

The collection graph is a simple graph implementation that wraps a collection of triples. This class is intended to be used in places where a graph is required but iteration is the only expected operation. All graph operations are supported by CollectionGraph, but many are not efficient and will be slow on large collections. In cases with large collections a memory based graph will be more efficient, for example the class GraphMem. [Jen] Class CollectionGraph

3.4 QueryVisitor extends ElementVisitorBase

The QueryVisitor class is intended to visit a partial user query and extract the basic graph pattern without the unnecessary filters. QueryVisitor extends ElementVisitorBase, this base class is an ElementVisitor that does nothing. Derived classes must override its methods, in order to do stuff. It saves writing lots of empty visit methods, when only interested in a few element types. The derived class QueryVisitor is visiting ElementPathBlocks, an ElementPathBlock basically represents the WHERE block of a SPARQL query, a SPARQL basic graph pattern. [Jen] Class ElementPathBlock] How to set the WHERE block when building or modifying a partial query can be seen in the section for the PartialQuery class. [Jen] Class ElementVisitorBase
3.5 PartialQuery extends Query

@override
public void visit(ElementPathBlock pathBlock)
{
    Iterator<TriplePath> ittp = pathBlock.patternElts();
    while (ittp.hasNext()) {
        TriplePath tp = ittp.next();

        if (tp.isTriple()) {
            whereBlock.addTriple(tp.asTriple());
        }
    }
}

QueryVisitor overrides the method, visit, from ElementVisitorBase, for the purpose of looking through ElementPathBlocks and making WHERE blocks out of them. Making WHERE blocks without the filters, is done by iterating over the triple paths of the block, and checking if they actually are triples, discarding the ones that are not. Triple paths that are not triples can be filters, basic pattern groups or sub queries for example. OptiqueVQS does not make extra basic graph patterns grouping, otherwise one would have to check begin a new visit to that path, iterating further.

public ElementPathBlock extractBasicPattern(Query query) {
    QueryVisitor visitor = new QueryVisitor();
    ElementWalker.walk(query.getQueryPattern(), visitor);

    return visitor.whereBlock;
}

When having a derived class of ElementVisitorBase, one walks the ElementPathBlock of the original query with the help of the class ElementWalker. The element path block from the original query you get with the "getQueryPattern" method. ElementWalker is an element visitor that walks the graph pattern tree, the where block for one query level. To process sub queries and EXISTS filters, one needs to call down these themselves by the visitor. [Jen] Class ElementWalker

3.5 PartialQuery extends Query

The class PartialQuery has the role of representing the end-users partial query, that is modified to extract seen extensions on the basic query graph. The PartialQuery class extends the class Query, and can then be use directly with QueryExecutionFactory to query models or RDFConnection’s query method. When querying or updating through an RDF connection an “SparqlQueryConnection” is used, when using the graph store protocol RDFDatasetConnection is used. [Jen] Interface RDFConnection]
3.5. PartialQuery extends Query

RDFConnection provides a unified set of operations for working on RDF with SPARQL operations. It provides SPARQL Query, SPARQL Update, and the SPARQL Graph Store operations. RDFConnection applies to both local data and remote data using HTTP and the SPARQL protocols. The Txn class, for transactions, is the preferred way to work with RDF data. Operations on an RDFConnection outside of an application-controlled transaction will cause the system to add one for the duration operations. This "autocommit" feature may lead to inefficient operations due to excessive overhead. [Jenz, RDFConnection: SPARQL operations API]

Now let's look at how the PartialQuery class manipulates the original query, extracting its WHERE block's basic graph pattern and adding it to itself. How it programmatically sets the query FORM with DISTINCT and the projection variables. How it makes the part that matches the extensions and how it makes the special FILTER. Then at the end, how all the parts are set together forming the modified partial query ready for use.

When working with SPARQL in the Jena framework, the go-to API to use is the ARQ API. In many other semantic web applications one will find that working with static queries are restrictive. Manipulating the partial query just by simply diving directly into the query string, changing values, adding the filter, and such by string manipulation is a fraught process. Syntax error await, and in the long run doing query manipulation with the ARQ API is way easier. [Jenab, Manipulating SPARQL with ARQ]

Since the PartialQuery inherits from Query, one can use all its methods to modify the current instance of PartialQuery that is being created with parts from the original query. The prefix mapping of the original query, meaning all the namespaces and their pseudonym, are taken from the original query and used as is. The basic query pattern from the original query is visited with a QueryVisitor, and the FILTER clean basic graph pattern is set as the modified partial query's query pattern.

```java
setPrefixMapping(originalQuery.getPrefixMapping());
ElementPathBlock whereBlock =
    new QueryVisitor().getBasicPattern(
        originalQuery.getQueryPattern())
setQueryPattern(whereBlock);
```

The modified partial query needs to SELECT the three projection variables binding the extensions. These projections variables are represented by the class Var and are allocated through the Var class's static method alloc. These are to be used around in the one added triple to be added, the OPTIONAL, the FILTER and as the projection variables to be set. [Jenu, Class Var]

```java
Var prop = Var.alloc("prop");
Var cons = Var.alloc("cons");
Var type = Var.alloc("type");
```
PartialQuery makes an ElementGroup \([\text{Jenk}]\) Class ElementGroup] to hold the elements; triple block element \([\text{Jenn}]\) Class ElementTripleBlock] and the optional element \([\text{Jenl}]\) Class ElementOptional]. The classes ElementTripleBlock and ElementOptional are both of the class Element. \([\text{Jeni}]\) Class Element\] The triple and optional is add to the where block ElementPathBlock. This element group will be set as the query pattern.

```java
ElementGroup group = new ElementGroup();
whereBlock.addTriple(Triple.create(pivot, prop, cons));

ElementTriplesBlock optional = new ElementTriplesBlock();
optional.addTriple(Triple.create(cons, RDF.type.asNode(), type));
ElementOptional optionalBlock = new ElementOptional(optional);
group.addElement(whereBlock);
group.addElement(optionalBlock);
```

For the creation of the special FILTER represented by an element filter \([\text{Jenj}]\) Class ElementFilter] with the given expression represented with an Expr instance. The Expr interface has a whole bunch of implementing classes for creating all kinds of expressions. \([\text{Jenv}]\) Interface Expr] The classes used here to create the expression for the filter, was E.StrStartsWith and E.Str. The variable "?cons" is turned into an instance of ExprVar, in order to use it as a component in the expression. \([\text{Jenq}]\) Class ExprVar]

```java
Expr filter;
ElementFilter filterBlock;
filter = new E.StrStartsWith(new E.Str(new ExprVar(cons)),
new NodeValueString(QueryModel.instantPrefix));
filterBlock = new ElementFilter(filter);
group.addElement(filterBlock);
```

The QueryModel instant prefix is "_:q". The class "NodeValueString" implements Expr, and is need to turn a string into a part of the expression. The class "E.Str" makes it clear that we want the to do something with the string representation of the node that is bounded to the variable "?cons" in each match.

```java
setQueryPattern(group);
setQuerySelectType();
setDistinct(true);
addResultVar(cons);
addResultVar(prop);
addResultVar(type);
```

Lastly the PartialQuery class sets the query pattern that has been modified. It sets the query type to SELECT with the method "setQuerySelectType", and to set the query to select only DISTINCT matches, one must call "setDistinct"
3.6. QueryModel extends ModelCom

with the truth value true. The variables who’s bindings are selected are added
in the order you want them to appear in the query.

3.6 QueryModel extends ModelCom

The QueryModel represents the instantiated model of a query’s basic graph
pattern, that has gotten its variables made into resources. The QueryModel
can turn a query into a model, but also turning the model back into an element
path block.

```
Iterator<TriplePath> paths =
elementPathBlock.patternElts();
while (paths.hasNext()) {
    String tag = "_:q" + queryNumber;
    TriplePath tp = paths.next();

    if (tp.isTriple()) {
        String sub = tp.getSubject().toString().replace("?", tag);
        String pre = tp.getPredicate().toString();
        String obj = tp.getObject().toString().replace("?", tag);

        Resource subject =
            ResourceFactory.createResource(sub);
        Property predicate =
            ResourceFactory.createProperty(pre);
        Resource object =
            ResourceFactory.createResource(obj);

        Statement stmt = ResourceFactory.createStatement(
            subject, predicate, object);
        add(stmt);
    }
}
```

This transformation to a model is done visiting the element path block of
the original query.

```
ElementPathBlock block = new ElementPathBlock();
StmtIterator iterator = this.listStatements();
while (iterator.hasNext()) {
    Triple phi = iterator.next().asTriple();

    Node subject = phi.getSubject();
    Node predicate = phi.getPredicate();
    Node object = phi.getObject();
```
if (subject.toString().matches("_:q\d+.\*"))
    subject = Var.alloc(str.replaceFirst("_:q", ""));

if (object.toString().matches("_:q\d+.\*"))
    object = Var.alloc(str.replaceFirst("_:q", ""));

    block.addTriple(Triple.create(sub, pre, obj));
} return block;

This reversal of the triples in the model, turns the instantiated variables in the triples back into variables, making the triples patterns. They are added to an element path block that can then be used as a where block.
CHAPTER 4

Collaborative Ranker

This ranker implementation uses the non-distributed, non-hadoop-based recommender engine / collaborative filtering code inside Mahout. A Mahout-based collaborative filtering engine takes users preferences for items and returns estimated preferences for other items. Mahout provides top-level packages that define interfaces to key abstractions like DataModel, UserSimilarity, ItemSimilarity, UserNeighborhood, and Recommender. Subpackages of org.apache.mahout.cf.taste.impl hold implementations of these interfaces.

4.1 Recommender

Mahout’s recommenders expect interactions between users and items as input. Users in this case are the queries in the query log, and the items are all the different types of extensions seen throughout the whole of the query log. The easiest way to supply such data to Mahout is in the form of a textfile, more specifically a CSV file (Comma-Separated Values). The CSV file has the format userID, itemID, value. Here userID refers to a particular query given that ID, and itemID refers to a particular type of extension given ID. Values denotes the strength of the interactions, in this case it is either one or zero depending on if given extension is seen in the given query.

The class for this collaborative filtering based ranker also inherits from linked has map and implements the Ranking interface, can then given to the Experiment class that will perform the experiment in the experiment chapter. The first thing that this ranker has to do is creating the CSV file of the query log that is going to be given to Mahout. The filtering based ranker utilizes the classes from the ontology based ranker, to handle much of what it needs to do be done with the queries.

```java
public class FilteringBasedRanking extends LinkedHashMap<List<Resource>, Double>
   implements Ranking
```

The filtering based ranker makes a log model out of all the queries in a given query log. Then with this model it queries it for all the type of extensions that has ever been seen in the query log. The query below does this job of getting all
the types of extensions. The unique extension types of extensions are the items to be recommended by the different queries if they are used by them or not.

```java
public final String itemQueryString =
    "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> "
+ "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> "
+ "SELECT DISTINCT ?prop ?type "
+ "WHERE { "
+ "    ?x ?prop ?c . "
+ "    OPTIONAL { ?c rdf:type ?type } "
+ "    FILTER NOT EXISTS { "
+ "        FILTER(regex(str(?prop),
+ "            http://www.w3.org/1999/02/22-rdf-syntax-ns#type")) } "
+ "    } "
+ "};
```

When the filtering based ranker has all the types of extensions ever seen used, then it for each query in the query log, sees if the query contains which of the extensions. The query log has 97 queries, so if there where 100 different unique extensions, then the CVS file would contain 9700 entries, each query is given a user id and recommends each extension with a one or a zero, one if in the query. The user with ID zero is the end-user’s partial made query, which Mahout will be asked to give recommendations for. The end-user’s partial query will have few recommended entries since it not completed. The CVS file contain entries that looks like the ones below:

```
0,55,0
0,6,1
0,31,0
0,19,0
1,2,1
1,45,0
1,30,1
1,42,0
2,64,0
2,25,1
2,50,0
2,1,1
96,27,0
96,33,0
96,8,1
96,56,1
```

Given a query with the basic graph pattern like this:

``` RDF
?w a :Wellbore ;
:name ?wellbore ;
:hasWellboreInterval ?wi ;
:hasFormationPressure ?fp .
?fp a :FormationPressure ;
:hasDepthMeasurement ?fp_depth .
```
4.2. Implementation

The filtering based ranker will have to, for query/user list which extensions/items that that user recommending and which it does not.

<table>
<thead>
<tr>
<th>Property</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasWellboreInterval</td>
<td>1</td>
</tr>
<tr>
<td>hasMeasurement</td>
<td>TemperatureMeasurement 0</td>
</tr>
<tr>
<td>hasFormationPressure</td>
<td>FormationPressure 1</td>
</tr>
<tr>
<td>hasWellbore</td>
<td>0</td>
</tr>
<tr>
<td>hasDepth</td>
<td>0</td>
</tr>
<tr>
<td>hasDepthMeasurement</td>
<td>MeasuredDepth 1</td>
</tr>
<tr>
<td>hasWellboreInterval</td>
<td>StratigraphicZone 0</td>
</tr>
<tr>
<td>name</td>
<td>1</td>
</tr>
</tbody>
</table>

After the CVS file has been created it can be given to Mahout, and it will generate an extension probability ranking like this:

<table>
<thead>
<tr>
<th>Property</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>0.9941246510</td>
</tr>
<tr>
<td>hasUnit</td>
<td>0.8396007419</td>
</tr>
<tr>
<td>valueInStandardUnit</td>
<td>0.8217660785</td>
</tr>
<tr>
<td>hasWellboreInterval</td>
<td>0.586978599</td>
</tr>
<tr>
<td>hasTopDepthMeasurement</td>
<td>MeasuredDepth 0.5863978267</td>
</tr>
<tr>
<td>hasBottomDepthMeasurement</td>
<td>MeasuredDepth 0.4800274968</td>
</tr>
<tr>
<td>overlapsWellboreInterval</td>
<td>0.4709674120</td>
</tr>
<tr>
<td>hasWellbore</td>
<td>Wellbore 0.4098887010</td>
</tr>
</tbody>
</table>

A difference between the filtering based ranker and the ontology based ranker, is that the filtering based ranker will give probabilities for every extension, so the above ranking is just the first few of all the entries in the ranking. The ontology based ranker on the other hand, will only give ranking for extensions it sees on the same type of basic graph pattern, therefore the number of ranked extension can vary greatly.

4.2 Implementation

The interface called DataModel handles interaction data, meaning the interactions between users on items, and it parses the CVS file.

```java
DataModel model = new FileDataModel(cvsFile);
```

For finding similar users, the recommender looks at other users with the most similar recommendations, and pick the novel recommendations from their items. For comparing interactions, a common method is to compute the correlation coefficient between their interactions. The Pearson correlation coefficient is the covariance of the two recommendations divided by the product of their standard deviations.

```java
UserSimilarity similarity =
    new PearsonCorrelationSimilarity(model);
```
4.3 Miscellaneous

Next one must define which similar users to leverage for the recommender. Should every other user have a weigh in, or just the ones in the same ballpark as the partial query. Using the threshold of 0.1 works fine for this papers purposes, it excluding all users with a similarity less then 0.1.

```java
UserNeighborhood neighborhood =
    new ThresholdUserNeighborhood(threshold, similarity, model);
```

To now compute recommendations for a particular user, one must create a user-based recommender, it uses the datamodel, neighborhood and similarity created above.

```java
Recommender recommender =
    new GenericUserBasedRecommender(model, neighborhood, similarity);
```

For larger data models one might consider using a caching recommender, helping runtime performance. [Maha User-Based Recommender]

```java
Recommender cachingRecommender =
    new CachingRecommender(recommender);
```

```java
cachingRecommender.recommend(userId, limit, true);
```

4.3 Miscellaneous

The filtering based ranker must keep track of which item ID was used for which extension, and which user ID was used for which query in the query log. Using maps to keep track of these connections are one convenient way of doing this. To this purpose one might like to just reverse a map’s key and value roles, which can be done like this. [htta reverse map]

```java
Map<K,V> idToItemMap =
    itemToIdMap.entrySet().stream().collect(
        Collectors.toMap(
            Map.Entry::getValue, Map.Entry::getKey));
```

If you want to test things out with actual csv files, temporary files are a great way to go. You don’t need to care about where to put them or what to call them. [oraf Create Temp File]

```java
File temp = Files.createTempFile("", ".csv").toFile();
temp.deleteOnExit();
```
PART III

Evaluation
This experiment is being used to compare the ontology-based ranker with the collaborative filtering based ranker. It uses a query log of 97 queries, these queries are not made by OptiqueVQS. There are no real query logs yet made by OptiqueVQS at this time, no real use-cases to draw on, this example query log will have to make do. The experiment takes a query directory containing the queries making up the query log, an instance of the random class, that generates a stream of pseudorandom numbers [rand], Class Random], and one of the two rankers as seen through the interface Ranking.

```
public Experiment(String queryDirectory, Random random, 
                 Ranking ranking);
```

### 5.1 Setup

The experiment takes the query log and partitions it up into ten sets of randomly placed queries. What the experiment does next is to take each of these ten percents of the queries, one by one, and using them as partial queries on the rest of the ninety percent. So each query in the query log will be used as a partial query to query ninety percent of the query log, which is used for the log model.

For each query in the current partial query set, a random leaf node is removed, and the triple’s subject is used as the pivot, for the modification of the partial query. The triple’s predicate is used as the extension the end-users wants, and the extension that the rankings will have to try to get as high up the rankings as possible, in order to get a good score.

The score function gives 10 points if the extension is suggested first, then one less point for each place down, if the extension is tenth then 1 point is given, any place beneath that results zero points. The idea behind this, is the thought that an end-user would mostly benefit if the wanted extension was in the top ten at least. Otherwise the point of a ranker goes somewhat out the window, and the end-user still must look through a bunch of alternatives.

Here is the leaf node query string used to get the one type of extension that the experiment looks for in the rankings.
5.2 Details

First the QueryLog is made into an object, then it is partitioned into query logs of ten percent of the original.

```java
QueryLog log = new QueryLog(queryDirStr);

LinkedList<QueryLog> groups =
    partition(new QueryLog(queryDirStr), random);
```

For each set of ten percent of the original query log, the set is used as the partial query candidate set.

```java
QueryLog candidates = groups.removeFirst();
QueryLog queryLog = new QueryLog(restGroups);
```

Each query among the candidates, are used to query the combined set of the rest of the groups. A model of all the leaf nodes is made, and one of them is randomly selected.

```java
QueryModel queryModel = new QueryModel(query);
Model leafnodes = getLeafNodes(queryModel);
```

One random statement, a leaf node, is selected from the model of all the leaf nodes. This statement’s subject is to be used as the pivot and its predicate is to be used as the end-user’s wanted extension.

In order to compare the two ranking method’s average performance, this experiment is run twenty times on both rankings, and the scores from these runs are averaged out to see how much better the ontology-based ranker is doing against the collaborative filtering-based ranker. The experiment needs a minimum of two queries in the query log to function. The maximum number of points that a ranker can get, when 97 queries is used as in this case, is 970 points.
Statement extension =
    new RankerUtils().randomStatement(leafnodes, random);
queryModel.remove(extension);

The basic query pattern of the current query with the one statement removed, is being used as the end-user’s basic query pattern. The query model that has one less statement, is turned back into an element path block and used instead of the original query’s WHERE block. The reset of the original query, the pivot and the reduced query model is used to make the modified partial query. The pivot variable found in the removed statement, has been instantiated in the query model, it is turned back into a variable.

Var pivot = Var.alloc(
    extension.getSubject().getLocalName().replaceFirst(
        instancePrefix, ""));
PartialQuery partialQuery =
    new PartialQuery(query, pivot, queryModel);

The ranking is then populated, and the score is updated. When reusing a single ranking instance one must removed the previous entries. When scrubbing a ranking, one must not fall into the trap of removing entries over what you yourself is iterating over it. To avoid getting a concurrent modification exception, the Iterator’s own remove method can be used.

ranking.makeRanking(queryLog, partialQuery);
score += inTopTen(ranking, extension);
scrubRanking(ranking);

groups.addLast(candidates);

The last thing that was done above, was to reinsert the partition that one has finished unseeing as the set of partial query candidates. The algorithm is repeated ten times to do the same with all ten partitions on the rest of the ninety percent.
CHAPTER 6

Findings

6.1 Expectation

The findings one would possibly expect is that the ontology-based ranker is going to perform badly many of the times, and very well some of the times. This expectation would be because the basic ontology-based ranker only matches extensions on the exact basic graph patterns of the partial query. Since the experiment reduces the queries it uses as partial queries with only one leaf node, there are not many queries in the query log of 97 queries that matches. In this setup the expectation for the average performance is that the filtering-based one will be better, it does not have that many possible extensions to filter, and query log provides some user overlap.

6.2 Data Plots

Both of these scatter plots below, have a y-axis with a minimum score value of five hundred, and a maximum score value of seven hundred. All the scores from both rankers fit into this range. The x-axis of both are from one too twenty, representing the iteration number, each being one run though of the experiment.
Comparing the two different scatter plot, one sees that the ontology-based ranker’s scores does not vary as much as the filtering-based ranker’s scores. One could reason that this is because of the filtering-based ranker’s statistical approach, that would get somewhat more stable if the query log was larger. Since all the queries are used as a partial query to query over the other ninety percent, the only factors that varies is if the relevant queries for a partial query ends up in the ten percent with the partial query, or the ninety percent. The factor that varies is which random extension is removed form the partial query to be scored for, but since only leaf nodes are removed, the choice may not vary between that many alternatives for a given query.
The average between the two rankers is that the ontology-based ranker gets a score of approximately 0.83, of that of the filtering-based ranker. For each experiment number, the same random seed was given to the random generator, given to both ranking methods. The best average that the ontology-based ranker achieved, in single experiments compared together, is in the experiment numbered 15, where the ontology-based ranker achieved approximately 0.91 of that of the filtering-based ranker. The worst it did was iteration number 8, where it achieved approximately 0.781 of that of the filtering-based ranker.

### 6.3 Special Case

The cases where the ontology-based ranker outclasses the filtering-based ranker is where the partial query’s basic pattern is seen in few places in the query log, and some of those examples have the wanted extension. The partial query and wanted extension below is an example of the ontology-based ranker having a real advantage. The partial query:

```prefix
PREFIX : <http://purl.org/net/grafli#>
```

```sql
SELECT DISTINCT ?cons ?prop ?type
WHERE
  ?w :hasWellboreInterval ?int .
  ?col a :ChronoStratigraphicColumn .
}
```
6.3. Special Case

```
?w :name ?wellbore .
?well :locatedIn ?pos ;
  :hasWellbore ?w .
?f a :FluidZone .
?w a :Wellbore .
?int :hasUnit ?c1 .
?int a :StratigraphicZone .
?w :hasWellboreInterval ?f .
?f ?prop ?cons
OPTIONAL
  { ?cons a ?type }
  FILTER strstarts(str(?cons), "_:q")
}

The wanted extension:

[ _:q1732f, http://purl.org/net/grafli#fluidZoneContent,
  _:q1732content ]

Ontology-based Ranking Score: 10

1. fluidZoneContent 0.50
2. overlapsWellboreInterval StratigraphicZone 0.50

Filtering-based Ranking Score: 0

1. name 0.85
2. longitude 0.81
3. latitude 0.81
4. locatedIn 0.81
5. hasWellbore Wellbore 0.81
6. hasUnit 0.69
7. hasWellboreInterval 0.57
8. valueInStandardUnit 0.56
9. overlapsWellboreInterval 0.35
10. hasTopDepthMeasurement MeasuredDepth 0.35
...
18. hasDepthMeasurement MeasuredDepth 0.13
19. ofStratigraphicColumn ChronoStratigraphicColumn 0.11
20. fluidZoneContent 0.11
21. hasTopDepthMeasurement TrueVerticalDepth 0.09
22. hasFormationPressure FormationPressure 0.08
...
70. cgType 0.00
71. hasDepthMeasurement TrueVerticalDepth 0.00
```
6.3. Special Case

In the ontology-based ranking, the wanted extension "fluidZoneContent" ended up at first place, of a ranking with only two entries. The filtering-based ranker ends up suggesting the wanted extension on the twentieth place, basically having droned it having to consider all possible extensions.
CHAPTER  7

Conclusion

To conclude, the findings indicate that competitiveness of the basic ontology-based ranking method depends a lot on the query log available, but on average one cannot say that it perform better then the collaborative filtering based ranking method.

Exceptions were met, but with these findings there a couple of things to keep in mind. That the query log was not made with OptiqueVQS. That the query log available to experiment on only had 97 queries. That the rankings were scored on the place meant of only one of the two types of extension types that OptiqueVQS makes, and that only the leaf occurrences of this one type was taken to make the partial queries.

7.1 Similar Work

Most work that has gone into extension suggestion when formulating queries, has been in the context of SQL. This is not a formal semantics type of setting, so not much of that will help in developing an improved version of the ontology-based ranker. The paper "Towards Exploiting Query History for Adaptive Ontology-Based Visual Query Formulation" [Soy+14, 4 Related Work] mentions SnipSuggest, context-aware autocompletion for SQL, as an example of similar work. [Kho+10, SnipSuggest]

7.2 Way Forward

Improved versions of the ontology-based ranker could be made to use semantic similarity measures to weigh in on the probabilities of extensions, as the paper "Towards Exploiting Query History for Adaptive Ontology-Based Visual Query Formulation" [Soy+14, 5 Discussion] suggests. One technique that might be used for this purpose, is to add extra queries to the query log, based on the originals queries in that log and the given partial query, in order to make the query log skewed.

But instead, what I would consider doing, is to make a ranker that combines both the ontology-based and filtering-based approach. This ranker would use the collaborative filtering based approach as its base, then use the ontology-based approach to exclude entries in the ranking that has not been seen on the given
basic graph pattern. If the pattern excludes too many entries, by not including any extensions, because it sees non used on the given basic graph pattern, then I would start reducing this basic graph pattern, making it match more queries in the log. How many is too many to exclude, is up for discussion.
Bibliography


[Conb] W3C - World Wide Web Consortium. *Basic Graph Patterns*. URL: [https://www.w3.org/TR/sparql11-query/#BasicGraphPatterns](https://www.w3.org/TR/sparql11-query/#BasicGraphPatterns).


[Cond] W3C - World Wide Web Consortium. *B Internet Media Type, File Extension and Macintosh File Type (Normative)*. URL: [https://www.w3.org/TeamSubmission/turtle/#sec-mediaReg](https://www.w3.org/TeamSubmission/turtle/#sec-mediaReg).


Bibliography


 qep [Foue] Eclipse Foundation. 3.2.2 Querying a repository. url: http://docs.rdf4j.org/programming/#_querying_a_repository.


<table>
<thead>
<tr>
<th>Bibliography</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mar</strong></td>
</tr>
<tr>
<td><strong>yum</strong></td>
</tr>
<tr>
<td><strong>dca</strong></td>
</tr>
<tr>
<td><strong>dcc</strong></td>
</tr>
<tr>
<td><strong>aja</strong></td>
</tr>
<tr>
<td><strong>jen</strong></td>
</tr>
<tr>
<td><strong>ccg</strong></td>
</tr>
<tr>
<td><strong>epb</strong></td>
</tr>
<tr>
<td>Bib reference</td>
</tr>
<tr>
<td>---------------</td>
</tr>
</tbody>
</table>
Bibliography

- Mahe: Apache Mahout. *What is Apache Mahout?* URL: [http://docs.rdf4j.org/rdf-tutorial/#_example_15_sparql_select_queries](http://docs.rdf4j.org/rdf-tutorial/#_example_15_sparql_select_queries)
- Oraa: oracle.com. *Class Files.* URL: [https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html](https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html)
- Orac: oracle.com. *Class LinkedHashMap<K,V>.* URL: [https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html)
- Orag: oracle.com. *Interface Path.* URL: [https://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html](https://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html)
- Orah: oracle.com. *Interface PathMatcher.* URL: [https://docs.oracle.com/javase/8/docs/api/java/nio/file/PathMatcher.html](https://docs.oracle.com/javase/8/docs/api/java/nio/file/PathMatcher.html)
pro

plu
protégé. Protégé Plugin Library. URL: https://protegewiki.stanford.edu/wiki/Protégé_Plugin_Library

pvi
protégé. Protégé Views. URL: https://protegewiki.stanford.edu/wiki/Protege4Views

wik
protégé. Protégé wiki. URL: https://protegewiki.stanford.edu/wiki/Main_Page

q4j
Eclipse RDF4J. 8.3 Example 15: SPARQL SELECT Queries. URL: http://docs.rdf4j.org/rdf-tutorial/#_example_15_sparql_select_queries

r4j
Eclipse RDF4J. About - Eclipse RDF4J. URL: http://rdf4j.org/about/

f4j
Eclipse RDF4J. Download - Eclipse RDF4J. URL: http://rdf4j.org/download/

d4j
Eclipse RDF4J. The RDF4J Server REST API. URL: http://docs.rdf4j.org/rest-api/

ahm
Ahmet Soylu. Ahmet Soylu - senior engineer at DNV GL. URL: http://www.ahmetsoylu.com/

Log

vqs

omh