Supporting Multi-Ontology Federated Queries

1. Introduction

The Semantic Web [1] has long been regarded as the next step in Web development. One of the major technologies associated with the Semantic Web is SPARQL [2], a query language for Semantic Web data. This project builds upon a previous project that implemented a SPARQL federation, an interface to make queries across several independent Semantic Web databases [3]. As SPARQL allows queries over Semantic Web data, it necessary uses ontologies, or formal representations of commonly used terms in a domain. Semantic Web databases, or SPARQL endpoints as they are known, may use different ontologies to store their data than the ones being used by the client. It is desirable that clients use their own ontologies without worrying about remote databases. The goal of this project is to enable clients to make queries in their ontologies and translate these queries into the ontologies used by SPARQL endpoints in the federation.

An independent module was constructed to provide this translation. Using external information to describe how concepts in different ontologies map to each other, it carries a “find-replace” procedure on the given SPARQL query. This process is complicated by the existence of prefixes, which are aliases for URIs and must also be replaced with the full URI during the translation procedure. The module is able to handle complicated SPARQL queries and works efficiently.
The user interface of the SPARQL federation interface, which had been constructed previously, was also improved. Some scripts were added to make testing the queries much simpler. In addition, XSL stylesheets were added so that SPARQL results returned from the federation would be displayed in a tabular format.

2. Background

The Semantic Web is a framework for making data more machine-understandable by providing metadata on how pieces of information on the Web are related. It works much like other relational models in computer science, such as those used by relational databases. A piece of data is a single entity, but may be related to other data in some way by some explicitly-defined relation.

Normally, these relations are represented as a graph, where the nodes represent actual data objects and the edges represent a relation. Resource Description Framework (RDF) is the standard language used to store Semantic Web data and establish the relations. These are represented as triples of the form “subject, predicate, object.” The subject is one piece of data, the predicate is the relation, and the object is another piece of data to which the subject is related. For example, we may have “‘Herman’ foaf:brother ‘Steve’.” This triple expresses that Herman has a brother Steve. The relations and objects come from ontologies, which are online data files that explicitly define objects and relations, and state what deductions may be made from these relations. For example, “foaf:brother” is a relation located in the Friend of a Friend (FOAF) ontology, which contains predicates to establish relations among people. The FOAF ontology is located online. To access the FOAF ontology, a Uniform Resource Identifier (URI), which operates much like a web link, must be included. The URI makes reference to where the ontology file is located. Often, instead of including long URIs, prefixes are defined as aliases for URIs. For example, the “foaf:” prefix used above is an alias for the URI
RDF graphs may be queried using SPARQL. It shares some of its syntax with SQL, the standard query language for databases. Here is an example SPARQL query, taken from the SPARQL Federator webpage [7]:

```
SELECT ?name ?job
WHERE {
}
```

The query asks the RDF database to return two variables, identified as “?name” and “?job.” These two terms may have nothing to do with an actual name or job, and are just variable names. These variables are constrained to match triples in the RDF graph satisfying the three constraints in the WHERE clause. Namely, there must be a person “p” which was the president of the United States (constraint three). Then, take the object of that person’s “foaf:name” and “dbpedia:occupation” and return them. This is a rule-based system and has the same syntax as a forward or backward chainer. Note that this query uses both prefixes, such as “foaf:” and “dbpedia,” and complete URIs such as in the last triple.
The Decentralized Information Group (DIG) at the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL) focuses on methods to improve SPARQL querying. One such way is by implementing a SPARQL federation. In a database federation, it is possible for a user to query several independent databases simultaneously by using an intermediate server. The client sends out a single query, and the proxy server uses pre-cataloged information about each of the databases to get the relevant data from that database. The server then aggregates the returned data into a single format and returns it to the client. Users are not aware that they may be querying several databases as once. The current SPARQL federation at DIG is accessible via a web interface [7]. The interface is created using a CGI script implemented in Python.

In SPARQL, there are several difficulties in implementing a federation. One arises from the use of ontologies. Users should be able to use a single ontology when writing SPARQL queries, without knowing what ontologies are used each of the databases. This is important both for usability, as it is quite confusing to keep track of which ontologies are being used, and is important for modularity, as the user should not know how data is formatted in the different SPARQL endpoints.

3. Approach

The main project had two main goals. The first and more important goal was to implement a functional translation module in Python. The translation module needed to take in a SPARQL query and a file which explicitly states which terms must be translated as an input and output the query with the appropriate translated namespaces. It was also necessary to handle terms that could not be translated appropriately. The translation should only affect the mapped terms, and should be able to support all types of SPARQL keywords.
The second goal was to improve the user interface in the current SPARQL federation website. This involved improving the form to make testing easier and to format SPARQL results into a readable format.

4. Design of Translation Module

4.1 Mapping Ontologies

The first thing that must be provided is a mapping ontology file. These are files written in RDF and explicitly specify a mapping from one term to another. An ontology known as the Base Mapping Ontology (identified with the prefix “mapont” in the following mapping ontologies) was defined [8] and describes how these mappings should be written. The Base Mapping Ontology currently contains only two terms. The first, “mappedTo,” is an RDF property. The second, “noMapping,” is an RDF class and is meant to be used as an object. There is currently no need to specify additional terms.
Below is an example of a simple ontology mapping file:

@prefix bio: <http://vocab.org/bio/0.1/> .
@prefix dbpedia: <http://dbpedia.org/ontology/> .
@prefix mapont: <http://dig.xvm.mit.edu/~yyyaron/mapOnt.n3> .

bio:position mapont:mappedTo dbpedia:occupation .

The file, written in n3 [9], specifies that the RDF property identified by “bio:position” should be replaced by the property defined by “dbpedia:occupation” (with the appropriate prefixes defined at the top of the file). It also specifies that the property “bio:health” cannot be appropriately mapped and should be handled accordingly. How these mappings are formed are beyond the scope of the project, and we assume that some external script has created an appropriate mapping from the user's ontology, which is used for querying, and the SPARQL databases' ontology used to describe the data.

4.2 SPARQL Translator

The translator module, which takes in a query and a mapping ontology, achieves its goal in several steps as described below.

4.2.1 Parsing the Mapping Ontology

The first step involves using ‘rdflib’ [10], which is a module that can parse RDF files, to parse the mapping ontology into a graph structure. The file must either be an RDF file or an N3 file. It is used to create two data structures. The first is a dictionary ‘mapping’ that stores pairs of terms ‘(a, b)’, specifying that term ‘a’ should be replaced with term ‘b’. The second is a set ‘unmappedSet’ which stores all terms that are not mapped. If we see a triple “a mapont:mappedTo b,” and b is “mapont:noMapping,” then we store b in unmappedSet. Otherwise, we store ‘(a, b)’ in mapping.
4.2.2 Parsing the Query

The query is parsed using the built-in Python regular expression module. The query is first split up by new line characters. This allows us to handle the query line by line. Then, each resulting line of the query is tokenized by whitespace. Finally, these tokenized lines may have empty string characters as elements of the list of terms. We therefore filter these out.

4.2.3 Parsing Namespaces

The script now begins a loop that will look through the topmost lines of the query. In particular, any line that begins with “PREFIX” specifies a URI that may possibly be mapped. We thus put all of these namespaces into a dictionary namespaces which contains pairs of strings (prefix, URI). Once a different keyword is seen (for example, the ‘SELECT’ or ‘CONSTRUCT’ keyword), we leave this loop.

4.2.4 Translation

We now begin to look through the actual query. For each term, we check to see if the term contains the prefix in namespaces as a key. If it does, then the prefix is replaced with the corresponding URI. This makes translation much easier, as ‘rdflib’ removes the prefixes from its terms and stores them as URIs.
Next, we look to see if the term is in ‘unmappedSet’. If it is, then the term has no correct mapping and we put the entire query line in a list ‘unmappableLines.’ The line is left out from the final query translation and is returned to the user separately. Otherwise, if the term is in the dictionary ‘mapping’, then we replace it with the corresponding translated term.

Figure 3: Flowchart representing how translation of individual terms is done.
4.2.5 Formatting

This step is used to create the final output. The tokenized lines are aggregated into a single string. New line and tab characters are inserted where appropriate to make the translated query more readable. The module thus returns two items. First, it returns the newly translated query excluding the unmapped lines. Second, it returns all lines that were not mappable in list format. It is up to the user to handle these.

5. Improving the Web Interface

5.1 Adding the Translation Module

The translation module is called in the federation script before sending the query to the databases. The original script using the query to retrieve data remains unchanged. Currently, the mapping file used in the web interface for the federation is named “federation_mapping.n3” and is identical to the mapping ontology provided in section A.1.

Any lines that did not have a mapping and could not be translated are printed separately, which informs the user that they were left out. The translated query is sent off to the federation, and the results are returned. In the future, it may be better to ask the user if the newly-translated query should be sent to the federation, as dropping some of the triples may make a query pointless. This may involve significantly altering the CGI script, however.

5.2 Other improvements

To improve the current SPARQL federation user interface, two additions were made. First, it was previously the case that users were allowed to pick test queries for the federation. These would not be shown unless the page reloaded, though. Using Javascript, the textbox containing the query to be sent was made dynamic, and its contents were updated when a test query is selected. This allows a user to
see the test queries before they are sent, making the interface more informative. The script used is rather simple. It stores the queries as strings which are inserted in the textbox on selecting a test query. Adding new test queries is also easy, though writing them is slightly complicated as they must be printed as Python strings that will match appropriate Javascript strings.

Next, the SPARQL results were being returned as HTML (Hyper Text Markup Language) without formatting. Using an XSLTProcessor [10], a construct native to the Firefox web browser, and a pre-defined XSL-stylesheet, which formats SPARQL-xml, the results were arranged in tabular format. To achieve this, the results of the query were first written on an intermediate file, temp_xml_query.xml. This was then fed into the XSLTProcessor with the XSL file. The result was appended to the end of the CGI-produced webpage. While storing the contents in an intermediate file is not a particularly elegant solution, it avoids needing to directly imbed styles within the SPARQL results and hence is much simpler to implement.

![Figure 4: Sample Formatted SPARQL Results](image-url)
6. Analysis

6.1 Quality of Translation

The translation module is able to consistently translate both simple and complex queries. As it only modifies terms with prefixes, it does not modify most parts of the query. As a result, queries containing FILTER, GRAPH, ASK, or other uncommon terms will be translated correctly. Unit testing consisted of using ten queries of different types. Four of them were taken from the SPARQL federation website. The others were either made up or taken from various locations. See the Appendix for results of translation, including the contents of the mapping file.

As the parsing is dependent on formatting, there are some undesirable effects in translation. For example, if triples are not placed on new lines, and one triple contains a prefix that is not mappable, then all the triples will be taken out of the query. This would probably change the query greatly, and may make it nonsensical. It may be better in the future to remove triples from the query instead of an entire line. This, however, would involve finding a good way to split a query up based off triples. It would also not allow us to give feedback to the user regarding where the unmappable term lies.

6.2 Runtime and Possible Optimizations

There are several factors affecting the runtime. First, parsing the mapping file depends on the rdflib library. In addition, using the regular expression module ‘re’ depends on the size of the query. We assume both take linear time with respect to the input, though this may be incorrect. Then, we iterate over how many terms are in the query, and for each term we iterate over the number of namespaces in the query. Therefore, if $t$ is the number of terms in the query, and $m$ is the number of namespaces provided in the header, then the theoretical complexity is $O(mt)$ (ignoring the ‘rdflib’ and ‘re’ functions).
The query would have to get extremely large for the runtime to get significantly noticeable. Since most queries are relatively short, though, this should not be a problem. If it does become a problem, looping over the namespaces for each term in the query probably takes the most amount of time in the program. If the number of prefixes in the query and the terms are about the same, then the runtime could become quadratic with the size of the query. One optimization that could be made is to explicitly find a prefix in a term and simply use the dictionary to replace it. However, this slightly complicated, as it could introduce several unforeseen edge cases. For example, both URIs and prefixes use the ‘:’ character. Doing a simple split based off ‘:’ would be erroneous. This optimization is very possible to implement, though, if it is necessary. However, it is probably much more important to look at how using the Python regular expression module and the ‘rdflib’ module operate if runtime becomes a problem.

7. Conclusion and Future Work

The translation module is able to translate SPARQL queries from one ontology to another with the use of a mapping file. The translation leaves most of the original query intact, and simply replaces terms that should be mapped. Those terms which cannot be replaced are excluded from the translation and the client is informed of these un-translatable terms. It is up to the client to handle the new query and the un-translated terms accordingly. In the DIG web interface, the new query is still sent to the federation, and the un-translated terms are printed to the user.

Overall, the module works extremely well. It translates queries correctly and rapidly. There is room for improvement, however. First, as mentioned, the formatting of the SPARQL query combined with the un-translated terms may have an adverse effect on what the user receives. Future work may want to reconsider how to handle un-translated terms, or expand the current functionality to eliminate disjoint
triples or conditions.

There are minor problems with formatting the translated query, mostly because not all keywords were taken into account. Future iterations may wish to look for a way to format the query without relying on the use of specific keywords.

The mapping module assumes that the mapping file has been constructed. While actually constructing the mapping is done elsewhere, it may be worthwhile to make sure that the file abides by the format needed by the translation module. In addition, no checking is done when parsing the mapping file. It would be useful to introduce some mechanisms to verify that the file is formatted properly.

While I did not manage to find a SPARQL parser, one might exist that is compatible with rdflib. It may be simpler and faster to use it to translate the query.

The functionality of the current translation module is rather straightforward. Terms are mapped in an external ontology, and then simply replaced in the query. One could imagine much more flexible options for translation. Suppose the term “foaf:name” could be mapped to two different terms, for example. Then it may be worthwhile to allow the option of creating two new triples from one triple. Future research in translating ontologies should consider these possibilities.

8. Acknowledgments

I would like to thank Lalana Kagal for aiding me in formulating this project and advising me, and Matthew Cherian for his work in creating the SPARQL federation.
9. References


<http://www.w3.org/TR/rdf-sparql-query/>.


<http://www.w3.org/TR/rdf-syntax/>.


<http://www.w3.org/2000/10/swap/Primer>.


[11] XSLT Processor Documentation,
A. Appendix

A.1 Test Results

Below are the test cases for the translation module, along with the corresponding translation. Most of the output should be unchanged from the input. Mapped terms should be replaced in the final query, and unmapped terms should be printed at the end of the output, apart from the actual query. Those test queries taken from elsewhere are referenced and may be found in the references section. These tests may be run by running the file located at web.mit.edu/yyyaron/Public.mapQuery.py. It is also necessary to download the file located at web.mit.edu/yyyaron/Public/federated_ontology.n3 and make sure it is in the same folder.

The ontology mapping used for testing has the following contents:

@prefix bio: <http://vocab.org/bio/0.1/> .
@prefix dbpedia: <http://dbpedia.org/ontology/> .
@prefix mapont: <http://dig.xvm.mit.edu/~yyyaron/mapOnt.n3> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

bio:position mapont:mappedTo dbpedia:occupation .
bio:age mapont:mappedTo mapont:noMapping .
bio:health mapont:mappedTo foaf:living .
foaf:surname mapont:mappedTo mapont:noMapping .

A.1.1 Test 1

Input:

SELECT ?v WHERE { ?v ?p 42 . ?v ?n "Yotam"}

Output:

SELECT ?v
WHERE
{ ?v ?p 42 .
  ?v ?n "Yotam"}

A.1.2 Test 2

Input:

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
  ?person foaf:name ?name .}
A.1.3 Test 3 [2]
Input:
```sparql
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE {
  ?person foaf:name ?name .
} LIMIT 50
```

Output:
```sparql
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE {
} LIMIT 50
```

A.1.4 Test 4 [2]
Input:
```sparql
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox WHERE {
  ?x foaf:name ?name .
  FILTER regex(?name, "Smith")
}
```

Output:
```sparql
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox WHERE {
  ?x <http://xmlns.com/foaf/0.1/name> ?name .
  FILTER regex(?name, "Smith")
}
```

A.1.5 Test 5 [2]
Input:
```sparql
PREFIX data: <http://example.org/foaf/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?mbox ?nick ?ppd
```
FROM NAMED <http://example.org/foaf/aliceFoaf>
FROM NAMED <http://example.org/foaf/bobFoaf>
WHERE
{
  GRAPH data:aliceFoaf
  {
    ?alice foaf:mbox <mailto:alice@work.example> ;
    foaf:knows ?whom .
    ?whom foaf:mbox ?mbox ;
    rdfs:seeAlso ?ppd .
    ?ppd a foaf:PersonalProfileDocument .
  }.
  GRAPH ?ppd
  {
    ?w foaf:mbox ?mbox ;
    foaf:nick ?nick
  }
}

Output:

PREFIX data: <http://example.org/foaf/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?mbox ?nick ?ppd FROM NAMED
  <http://example.org/foaf/aliceFoaf> FROM NAMED
  <http://example.org/foaf/bobFoaf>
WHERE
{
  GRAPH <http://example.org/foaf/aliceFoaf> { ?alice
    <http://xmlns.com/foaf/0.1/mbox> <mailto:alice@work.example> ;
    <http://www.w3.org/2000/01/rdf-schema#seeAlso> ?ppd .
    ?ppd a <http://xmlns.com/foaf/0.1/PersonalProfileDocument> .
  }.
  GRAPH ?ppd { ?w <http://xmlns.com/foaf/0.1/mbox> ?mbox
    <http://xmlns.com/foaf/0.1/nick> ?nick } }

A.1.6 Test 6 [7]

Input:

#Paris born Movie Stars

PREFIX dbpedia: <http://dbpedia.org/ontology/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?name ?m
WHERE {
}

Output:
#Paris born Movie Stars
PREFIX dbpedia: <http://dbpedia.org/ontology/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?name ?m
WHERE {
}

A.1.7 Test 7 [7]

Input:
#German Musicians who were born in Berlin

PREFIX dbpedia: <http://dbpedia.org/ontology/>
PREFIX dbp_resource: <http://dbpedia.org/resource/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dc_terms: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?n ?b ?label WHERE {
}

Output:
#German Musicians who were born in Berlin
PREFIX dbpedia: <http://dbpedia.org/ontology/>
PREFIX dbp_resource: <http://dbpedia.org/resource/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dc_terms: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?n ?b ?label
WHERE
  ?p <http://dbpedia.org/ontology/birthPlace>
  <http://dbpedia.org/resource/Category:German_musicians> .
  <http://dbpedia.org/resource/Category:German_musicians>
  }

A.1.8 Test 8 [7]

Input:

PREFIX bio: <http://vocab.org/bio/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dc_terms: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?name ?job ?age
WHERE
  ?p dc_terms:subject
  }

Output:

PREFIX bio: <http://vocab.org/bio/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dc_terms: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?name ?job ?age
WHERE
  ?p <http://purl.org/dc/terms/subject>
The line ?p <http://vocab.org/bio/0.1/age> ?age . could not be translated because < http://vocab.org/bio/0.1/age > does not have a proper mapping. This line has been dropped from the query.

A.1.9 Test 9 [7]

Input:
#People who played professional baseball and basketball

PREFIX dbpedia: <http://dbpedia.org/ontology>
PREFIX dbpedia_category: <http://dbpedia.org/resource/Category:>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dc_terms: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?name
WHERE {
}

Output:
#People who played professional baseball and basketball
PREFIX dbpedia: <http://dbpedia.org/ontology>
PREFIX dbpedia_category: <http://dbpedia.org/resource/Category:>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dc_terms: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?name
WHERE {
}
A.1.10 Test 10 [2]

Input:

```dot
PREFIX foaf: <http://xmlns.com/foaf/0.1/>  
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>

CONSTRUCT { ?x vcard:N _:v .  
            _:v vcard:givenName ?gname .  
            _:v vcard:familyName ?fname }
WHERE
{}  
```

Output:

```dot
PREFIX foaf: <http://xmlns.com/foaf/0.1/>  
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>

CONSTRUCT { ?x <http://www.w3.org/2001/vcard-rdf/3.0#N> _:v .  
            _:v <http://www.w3.org/2001/vcard-rdf/3.0#givenName> ?gname .  
            _:v <http://www.w3.org/2001/vcard-rdf/3.0#familyName> ?fname }
WHERE
{}  
```

The line `{ ?x <http://xmlns.com/foaf/0.1/surname> ?fname } UNION { ?x <http://xmlns.com/foaf/0.1/family_name> ?fname } .` could not be translated because `<http://xmlns.com/foaf/0.1/surname>` does not have a proper mapping. This line has been dropped from the query.
A.2 Code

Below is the code for the ‘translate’ function. The code is also available at web.mit.edu/yyyaron/Public.mapQuery.py.

```python
def translateQuery(query, mappingFileName):
    
mappingEqual = "http://dig.xvm.mit.edu/yyyaron/mapOnt.n3mappedTo"
unmapped = "http://dig.xvm.mit.edu/yyyaron/mapOnt.n3noMapping"
filename = mappingFileName

    #These are the keywords used to find the separation from the header of
    a query and the actual query.
    keyWords = ["SELECT", "CONSTRUCT", "ASK", "DESCRIBE"]

    #Step 0: Use rdflib to parse the mapping ontology.
    graph = Graph()
    if filename.endswith('.n3'):
        graph.parse(filename, format="n3")
    elif filename.endswith('.rdf'):
        graph.parse(filename)
    else:
        print "Please make sure the mapping file is either in n3 or rdf format."
        return 0

    #Step 1: Make a dictionary to store which terms in the mapping ontology
    should be mapped, and which terms are not mappable.
    mapping = {}
    unmappedSet = set()
    for t in graph:
        predicate = str(t[1].decode()) #Get string representation of
        relation
        if predicate == mappingEqual: #If we see mapont:mappedTo
            if str(t[2]) == unmapped: #If term is of class
                mapont:noMapping, add to unmapped set.
                unmappedSet.add(str(t[0]))
            else:
                mapping[str(t[0])] = str(t[2]) #Add mapping

    #step 2: Tokenize the query by new lines so we can work with the
    strings easily
    tokenizedLines = re.split('\n', query)
    for i in range(len(tokenizedLines)):
        tokenizedLines[i] = re.split('\s', tokenizedLines[i]) #Split into
        tokens
    tokenizedLines[i] = filter (lambda el: el != '', tokenizedLines[i])
    #Filter out whitespaces from list

    #step 3: Translate the namespaces in the query into URIs. If the URI is
    #to be mapped, then map it.
```
```
namespaces = {}
i = 0

while True:  # first loop, map prefixes in the header to namespaces
    if tokenizedLines[i] == []:  # Check necessary because of filter function above.
        i += 1
        continue
    if tokenizedLines[i][0] in keyWords:
        break  # Leave query
    if tokenizedLines[i][0] == 'PREFIX':  # Will have a namespace triple
        prefix = tokenizedLines[i][1]  # get prefix name
        uri = tokenizedLines[i][2]  # get uri corresponding to name
        # Take off the angle brackets from the uri
        namespaces[prefix] = uri[1:(len(uri) - 1)]
        i += 1

    # Non-PREFIX terms
    unmappableLines = []  # Storage for unmapped lines
    length = len(tokenizedLines)
    toPop = []  # List of lines that must be removed from query.
    while True:
        if i >= length:
            break  # Done with the query
        line = tokenizedLines[i]
        for j in range(len(line)):  # Go over all the terms in a line and see if it needs translating
            for s in namespaces:  # Check if term contains a prefix.
                if len(s) < len(line[j]) and s in line[j]:
                    base = line[j][len(s):]
                    line[j] = namespaces[s] + base
                    if line[j] in unmappedSet:  # Cannot be mapped. Make note of term and line and return to the user
                        unmappableLines += [(line[j], line)]
                        if not i in toPop:
                            toPop.append(i)

                        # Now, if the translated item is also in the mapping, translate.
                        if line[j] in mapping:
                            line[j] = mapping[line[j]]

                        # Finally, add angle brackets. If the term ends with a period, this should be handled accordingly
                        if line[j].endswith('.'):  # period
                            tokenlength = len(line[j]) - 1
                            line[j] = '<' + line[j][:-1] + '>'  # period
    return unmappableLines, toPop
else:
    line[j] = '<' + line[j] + '>'

    i += 1

    # Remove unmappable lines from tokenizedLines
    for i in toPop:
        tokenizedLines.pop(i)

    # Step 4: Aggregate lines. Newline and tab characters added for formatting final query.
    new = ''

    for line in tokenizedLines:
        for k in line:
            if len(k) == 0:
                continue
            if k == 'PREFIX' or k == 'WHERE' or (k in keyWords):
                new = new + '\n' # Put a newline before one of these terms
            new += k + " "
            if k.endswith(".")) or k.endswith(";")): # Done for readability purposes only
                new += "\n\t"
            if k == 'WHERE':
                new += '\n'

    return new, unmappableLines