A dataflow graph transformation language and query rewriting system for RDF ontologies*

Marianne Shaw[1], Landon T. Detwiler[2], James F. Brinkley[2,3], and Dan Suciu[1]

[1] Computer Science & Eng., University of Washington, Seattle WA
[2] Dept. of Biological Structure, University of Washington, Seattle, WA
[3] Dept. of Medical & Biological Informatics, University of Washington, Seattle, WA

Abstract. Users interested in biological and biomedical information sets on the semantic web are frequently not computer scientists. These researchers often find it difficult to use declarative query and view definition languages to manipulate these RDF data sets. We define a language IML consisting of a small number of graph transformations that can be composed in a dataflow style to transform RDF ontologies. The language's operations closely map to the high-level manipulations users undertake when transforming ontologies using a visual editor. To reduce the potentially high cost of evaluating queries over these transformations on demand, we describe a query rewriting engine for evaluating queries on IML views. The rewriter leverages IML's dataflow style and optimizations to eliminate unnecessary transformations in answering a query over an IML view. We evaluate our rewriter's performance on queries over use case view definitions on one or more biomedical ontologies.

1 Introduction

A number of biological and biomedical information sets have been developed for or converted to semantic web formats. These information sets include vocabularies, ontologies, and data sets. They may be available in basic RDF, a data model for the semantic web in which graphs are collections of triple statements, or languages with higher-level semantics, such as OWL. Researchers want to leverage the biomedical information available on the semantic web.

Mechanisms are available for scientists to manipulate RDF ontologies. Researchers can manually modify a copy of the content or develop a custom program to transform the data. Visual editors ([6][5][2]) can be used to modify and augment a copy of the data. Extraction tools such as PROMPT [21] can extract subsets of information; these subsets can be combined and modified by a visual editor. The high-level functionality of these visual tools maps well to researchers' mental model for transforming an ontology. Unfortunately, the data must be locally acquired and the user actions repeated when the data is updated.

Declarative view definition languages such as RVL [18], NetworkedGraphs [24], and vSPARQL [26] allow users to define views that transform an RDF information set. These view definitions can be evaluated on-demand, avoiding stale data

* This work was funded by NIH grant HL087706.

and expensive user actions. They can also be maintained, evolved, and used to define complex transformations. Unfortunately, non-technical users find it difficult to create declarative view definitions that perform the high-level operations available in visual editors. This problem is exacerbated by the need to reconstruct the unmodified data along with the transformations. Transformation languages enable users to specify only the modifications that need to be applied to data.

In this paper, we make two contributions. First, we present a high-level dataflow view definition language that closely matches users' mental model for transforming ontologies using visual editing tools. The language consists of a small number of graph transformations that can be composed in a dataflow style. Instead of a single declarative query, a user specified sequence of operations defines a transforming view over which queries can be rewritten and evaluated.

Second, we address the query processing challenge for our language. We present a query rewriting system that composes queries with IML view definitions, reducing the high cost of on-demand evaluation of queries on transformed ontologies. The rewriter leverages IML's dataflow style to eliminate transformations that are redundant or unnecessary for answering the query. The rewriter incorporates a set of optimizations to streamline the generated query. It then uses graph-specific statistics to produce an efficient query.

We evaluate our query rewriting engine on queries over a set of use case view definitions over one or more of four biomedical ontologies [26]. We compare the performance of our rewritten queries against the cost of first materializing and then querying a transformed ontology. 60% of the rewritable queries have an execution time at least 60% less than the view materialization time.

The paper is structured as follows. We present a motivating example in Sect. 2. Section 3 presents the InterMediate Language through an example. Section 4 presents the query rewriting engine and its optimizations. After describing our implementation in Sect. 5, we leverage IML definitions of nine use case views and their associated queries to evaluate our query rewriting engine's performance in Sect. 6. We discuss related work in Sect. 7 before concluding in Sect. 8.

2 Motivating Example

A radiologist spends significant time manually inspecting and annotating medical images. He notes normal anatomical objects and anomalous regions and growths; these annotations are used by a patients' doctor to suggest follow-up treatment.

To reduce the time and manual effort consumed by this task, the radiologist wants to develop an application (similar to [1]) that provides a list of medical terms for annotating an image. The radiologist will indicate the region of the body (e.g. gastrointestinal tract) that the image corresponds to, and the application should provide a set of terms that can be used to "tag" visible objects.

The radiologist needs the annotation terms that he associates with objects in the image to be well-defined. This ensures that there will be no misunderstanding by the doctors that read his findings. Therefore, the provided terms should be concepts from established biomedical references. The Foundational Model

of Anatomy (FMA) [23], a reference ontology modeling human anatomy, can be used for anatomical terms, and the National Cancer Institute Thesaurus (NCIt) [4], a vocabulary containing information about cancer, can be used to provide terms for anomalous growths.

A visual editor like Protege can be used to identify the objects that might be visible in a medical image. Simply displaying all terms is impractical; the FMA contains 75,000 classes and the NCIt contains 34,000 concepts. This identification process is tedious, time-intensive, and error-prone. For example, identifying visible parts of the gastrointestinal tract in the FMA requires inspection of more than 1300 objects. This manual process must be repeated every time the FMA and NCIt are updated – approximately yearly for the FMA, monthly for the NCIt. Instead, a view definition can be created and evaluated on demand.

3 A dataflow language for transforming RDF ontologies

We propose the InterMediate Language (IML), a view definition language that both 1) enables non-technical users to define transformations of RDF ontologies, and 2) makes it possible to efficiently answer queries over those transformations.

There are many ways to transform an ontology to create a new ontology. Facts about individual classes, properties, or restrictions can be added, deleted, or modified. Relevant subsets or subhierachies of the ontology can be extracted and combined, or restructured as needed. Unnecessary subsets or subhierarchies can be removed. Multiple ontologies can be merged to create a new one.

IML has been designed to support this range of functionality. The language consists of a small number of high-level graph transformations that correspond to the functionality provided in visual editors. (In Sect. 7, we compare IML to Protege/PROMPT, a popular visual ontology editor.) A sequence of operations produce a transforming view definition where output of one operation flows as input, via a default graph, to the next; this corresponds to making changes, sequentially, to a local copy of an ontology in a visual editor. Transforming view definitions are named and can be referenced by other transforming views.

3.1 IML Transforming Operations

IML contains selection, modification, addition, and utility operations. Table 1 contains the grammar for IML's operations. In their simplest form, IML operations operate on concretely specified resources. For more advanced transformations, IML leverages the syntax of SPARQL, the query language for RDF. IML uses the SPARQL grammar's `WhereClause` for querying graph patterns in an RDF graph and its `ConstructTemplate` for producing new graphs.

We discuss common IML operations in the context of a simplified IML view for the motivating example in Sect. 2. From the FMA, 1) `lines 1-7` extract the partonomy of the gastrointestinal tract, eliminating variants of the "part" property label; 2) `lines 8-11` identify the subclasses of `fma:Cardinal_organ_part` recursively (approximating visibility in images); and 3) `lines 13-15` extract the visible part hierarchy by joining the partonomy and visible organ parts.

Table 1. IML Transforming Operations

extract_edges ConstructTemplate OptClauses
extract_cgraph { *varOrTerm* ImlPropertyList } OptClauses
extract_reachable { *varOrTerm* ImlPropertyList } OptClauses
extract_path { *varOrTerm* ImlPropertyList *varOrTerm* } OptClauses
extract_recursive { ConstructTemplate OptClauses } { ConstructTemplate OptClauses }
add_edge < *varOrTerm verb varOrTerm* > OptClauses
delete_edge < *varOrTerm verb varOrTerm* > OptClauses
delete_node *varOrTerm* OptClauses
delete_property *verb* OptClauses
delete_cgraph { *varOrTerm* ImlPropertyList } OptClauses
replace_edge_property < *varOrTerm verb varOrTerm* > *verb* OptClauses
replace_edge_(subject|object|literal) < *varOrTerm verb varOrTerm* > *varOrTerm* OptClauses
replace_property *verb* verb OptClauses
replace_(node|literal) *varOrTerm varOrTerm* OptClauses
merge_nodes *varOrTerm*List CreateNode RetainElimList MergeSourceList OptClauses
split_node *varOrTerm* SplitNodeList OptClauses
union_graph SourceSelectorList
copy_graph SourceSelector
OptClauses := (**graph** SourceSelector)? WhereClause?
SourceSelector := *IRIref*
ImlPropertyList := '[' ImlProperty (',' ImlProperty)* ']'
ImlProperty := (*varOrTerm* |(**outgoing** | **incoming**) (*varOrTerm*)?)
CreateNode := **create** *SkolemFunction*
RetainElimList := (((**retain** | **eliminate**) *varOrTerm* ImlPropertyList)+)?
MergeSourceList := ((**merge_source** *varOrTerm sourceSelector*)+)?
SplitNodeList := CreateNamedNode RetainElimList (CreateNamedNode RetainElimList)+
CreateNamedNode := **create** *IRIref* ',' *SkolemFunction*
VarOrTermList := '[' *varOrTerm* (',' *varOrTerm*)* ']'

```
1) INPUT <http://.../fma>                                  # Extract partonomy of GI tract
2) { extract_cgraph { fma:GI_tract [outgoing(fma:regional_part), outgoing(fma:systemic_part),
3)       outgoing(fma:constitutional_part)] } graph <http://.../fma>
4)   replace_property fma:regional_part fma:part
5)   replace_property fma:systemic_part fma:part
6)   replace_property fma:constitutional_part fma:part
7) } OUTPUT <gi_part_hierarchy>

8)  INPUT <http://.../fma>        # Find subclasses of Cardinal_organ_part (visible on images)
9)  { extract_reachable { fma:Cardinal_organ_part [incoming(rdfs:subClassOf)] }
10)       graph <http://.../fma>
11) } OUTPUT <visible>

12) INPUT <gi_part_hierarchy>, <visible>  # Select visible parts by joining the part hierarchy
13) { extract_edges { ?a fma:part ?c }    # and the set of visible elements.
14)       where { graph <gi_part_hierarchy> { ?a fma:part ?c } .
15)         graph <visible> { ?x localhost:reaches ?c } . }
16)   add_edge <fma:Appendix fma:part fma:Appendix_tip>
17)   delete_edge <fma:Appendix fma:part fma:Tip_of_appendix>
18) } OUTPUT <visble_hierarchy>
```

Selection A technique used to create a new ontology is to select subsets of relevant information from an existing one and combine them, via join or union. The ability to extract relevant parts of an ontology is critical because many ontologies contain considerably more information than is needed by a scientist.

IML's `extract_edges` operation enables the selection of specific relevant edges from an ontology. For example, `extract_edges { ?a fma:FMAID ?c } where { ?a fma:FMAID ?c }` selects all of the `FMAIDs` from the FMA. The operation can also be used to join two RDF graphs. `Lines 13-15` extract the visible parts of the GI tract by joining the part hierarchy with the visible organ parts.

Commonly, users need to select an entire hierarchy from an ontology. Using the `extract_cgraph` operation, a user specifies a starting resource and a set of properties that are recursively followed to extract a connected graph. `Lines 2-3` select the partonomy of the gastrointestinal tract by recursively extracting all of its regional, systemic, and constitutional parts. Similarly, `extract_reachable` produces a list of all nodes that can be reached by recursing over a set of properties. `Line 9` recursively finds all of the subclasses of `fma:Cardinal_organ_part` to identify anatomical elements that may be visible on a medical image.

Modification When leveraging an existing ontology, scientists often delete, modify or rename some of the content. The `delete_edge` operation can be used to eliminate edges from an ontology; `line 17` deletes the edge specifying that `fma:Appendix` has part `fma:Tip_of_appendix`. All instances of a resource or property in a graph can be eliminated with `delete_node` or `delete_property`.

Users may need to modify triples from an RDF graph. The `replace_node` and `replace_property` operations replace all instances of a resource in a graph. `Lines 4-6` replace all extracted part edges in the GI tract partonomy with a uniform `fma:part` edge. For more fine-grained replacements, the `replace_edge_*` operations can change the subject, property, or object of a RDF triple.

Addition Users can add new facts to an ontology using IML's `add_edge` operation, which adds new triples to an RDF graph. `Line 16` adds a new edge indicating that `fma:Appendix` has part `fma:Appendix_tip`.

4 IML Query Rewriting and Optimization

A typical IML program is a dataflow diagram consisting of a sequence of IML operations. A naive approach to evaluate an IML program is to evaluate each operation sequentially. This is inefficient, as each operation produces an intermediate result that may be comparable in size to the input RDF data.

We have developed a system for rewriting queries over IML view definitions. The rewriter leverages IML's dataflow style to combine operations and eliminate transformations in the view definition that are unnecessary for answering the query, thus reducing query evaluation time.

The query rewriting engine is depicted in Fig. 1. The IML view definition and query are parsed into an abstract syntax tree. Individual operations are converted into a set of Query Pattern Rules (QPRs) and combined to create a rewritten QPR set representing the query. During this process, optimizations eliminate unnecessary or redundant transformations. Performance optimizations are applied before the query is converted to vSPARQL and evaluated.

vSPARQL is an extension to the SPARQL1.0 standard that enables transforming views through the use of (recursive) subqueries, regular-expression styled property path expressions, and dynamic node creation using skolem functions. We describe a vSPARQL view definition (below) for the IML view in Sect. 3.

vSPARQL supports `CONSTRUCT`-style subqueries to generate intermediate results. The subquery on `lines 14-17` creates an RDF graph `<visible>` listing all of the subclasses of `fma:Cardinal_organ_part` using a recursive property path

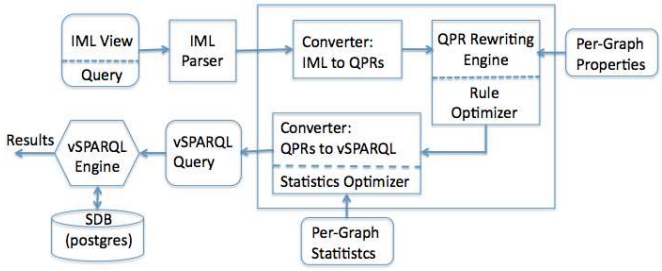

Fig. 1. Overview of IML query rewriting system.

expression. Recursive subqueries consist of two or more `CONSTRUCT` queries, a base case and a recursive case; the recursive case is repeatedly evaluated until a fixed point is reached. `Lines 2-13` define a recursive query that extracts the partonomy of the gastrointestinal tract into `<gi_part_hierarchy>`. The results of the two subqueries are joined on `lines 19-20`.

```
1) construct { ?a fma:part ?c . fma:Appendix fma:part fma:Appendix_tip }         # Add_edge
2) from namedv <gi_part_hierarchy> [                        # Extract partonomy of GI tract
3)   construct { fma:Gastrointestinal_tract fma:part ?c }                      # Base case
4)   from <http://.../fma>
5)   where { fma:Gastrointestinal_tract ?p ?c  .
6)     filter((?p=fma:regional_part)||(?p=fma:systemic_part)||(?p=fma:constitutional_part)) }
7)   union
8)   construct { ?prev fma:part ?c }                                          # Recursive case
9)   from <http://.../fma>
10)  from namedv <gi_part_hierarchy>
11)  where { graph <gi_part_hierarchy> { ?a fma:part ?prev }
12)    ?prev ?p ?c .
13)    filter((?p=fma:regional_part)||(?p=fma:systemic_part)||(?p=fma:constitutional_part)))}]
14) from namedv <visible> [    # Find subclasses of Cardinal_organ_part (visible on images)
15)   construct { fma:Cardinal_organ_part lcl:reaches ?b }
16)   from <http://.../fma>
17)   where { ?b rdfs:subClassOf* fma:Cardinal_organ_part }  ]
18) where {                          # Select visible parts by joining the part hierarchy
19)     graph <gi_part_hierarchy> { ?a fma:part ?c }     # and the set of visible elements.
20)     graph <visible> { ?x lcl:reaches ?c }
21)     FILTER((?a != fma:Tip_of_appendix) && (?c != fma:Tip_of_appendix))  # Delete_edge }
```

The working draft for SPARQL1.1 includes property path expressions, embedded subqueries, and skolem functions. Many, but not all, vSPARQL recursive subqueries can be expressed using property path expressions, including the subquery in `lines 2-13`. SPARQL1.1's property path expressions cannot require multiple constraints on nodes along a path, nor recursively restructure a graph.

4.1 Query Pattern Rule (QPR) Sets

During rewriting each IML operation is converted into a set of Query Pattern Rules (QPRs). Each QPR consists of a graph `pattern`, a list of `constraints`, and a graph `template`. The QPR indicates that if the constrained pattern (which may contain disallowed triples), is found in the graph to which it is applied, the corresponding `template` should be added to the output. A QPR is in Fig. 2.

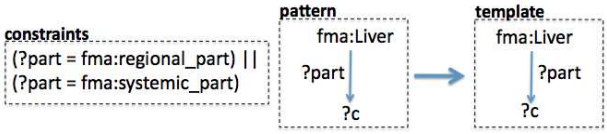

Fig. 2. Example QPR: If the default graph contains a triple matching (fma:Liver ?part ?c), where ?part is either fma:regional_part or fma:systemic_part, add it to our output.

For each IML operation, a *set* of QPRs is generated. The result of an IML operation is the union of all of the QPRs in the QPR set. Many operations permit the optional specification of a `WhereClause` for defining variables via query pattern bindings. The `WhereClause` is separated into graph pattern and `FILTER` elements; these define a QPR's `pattern` and `constraints`, respectively. `UNION` and `OPTIONAL` statements inside of `WhereClauses` cause multiple QPR sets to be generated, one for each combination of possible `WhereClause` patterns.

4.2 Query rewriting process

The rewriting engine starts with the last IML operation in the query and proceeds, operation by operation, towards the top of the query block. At each step, the IML operation is first converted into a QPR set; this QPR set is then combined with the working QPR set to produce a new working QPR set. If an IML operation `iml_op_x` references a subquery block via the `GRAPH` keyword, the rewriting engine recursively rewrites `iml_op_x`'s `pattern` over the named subquery block to produce a QPR set for `iml_op_x`. The rewriter combines the QPR set for `iml_op_x` with the current working set and continues to the next preceding IML operation. After the first IML operation in the query's subquery block is processed (i.e. the top of the IML block is reached), the working QPR set represents the overall rewritten query that must be evaluated.

As each IML operation `iml_op_x` is encountered, the rewriting engine combines `iml_op_x`'s QPR set with the working QPR set. It does this by combining *each* QPR in the working set with *each* of the QPRs in `iml_op_x`'s QPR set. If a QPR pattern has more than one element (i.e. triple pattern), each element is unified with each of iml_op_x's QPRs.

QPRs are combined by unifying the `pattern` of one QPR with the `template` of the other. If no unification is found, then no QPR is produced. If a unification is found, then the unifying values are substituted in to produce a new QPR.

Unification Example: We illustrate this process with an example. Figure 3 depicts two IML operations, `extract_edges` and `add_edge`, and their QPR sets. Each of `add_edge`'s two QPRs must be unified with `extract_edges`' single QPR. Unifying `add_edge`'s `QPR1.pattern` and `extract_edges`' `QPR1.template` produces a QPR equivalent to `extract_edges`' `QPR1`. Figure 3(c) presents the unification found for `add_edge`'s `QPR2.pattern` and `extract_edges`' `QPR1.template`, producing Fig 3(d)'s QPR. The final QPR set contains both of these QPRs.

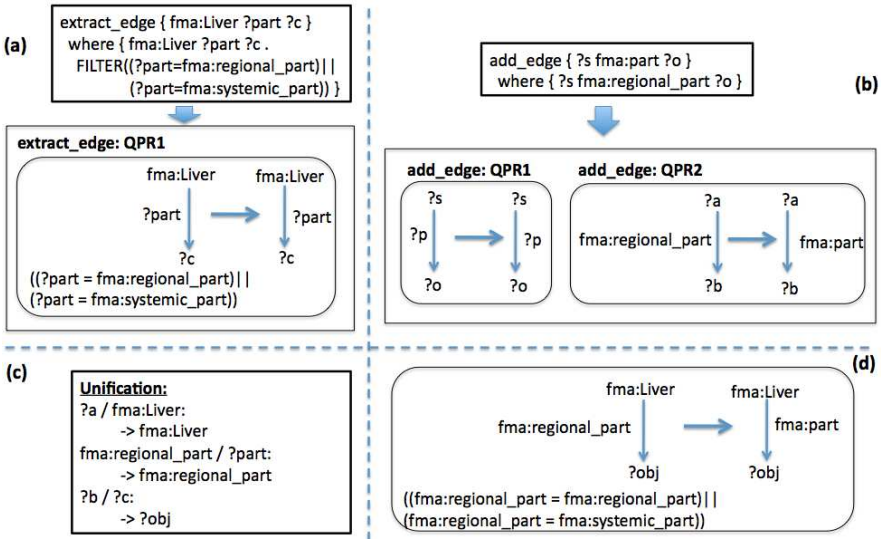

Fig. 3. Unifying (a) extract_edge's QPR1.template and (b) add_edge's QPR2.pattern produces the unification set (c) and QPR (d).

More generally, for two IML operations `iml_op_1` and `iml_op_2`, we combine the two QPRs by finding a possible unification of `pattern2` with `template1`. If a unification is found, we substitute those unifying values into `pattern1` and `constraint1` (producing `pattern1'` and `constraint1'`), and `constraint2` and `template2` (producing `constraint2'` and `template2'`) to generate a QPR (`result_QPR`) that can be added to the new working set.

```
iml_op_1: pattern1 + constraint1 -> template1
iml_op_2: pattern2 + constraint2 -> template2
result_QPR: pattern1' + constraint1' + constraint2' -> template2'
```

4.3 Rewriting optimizations

Rewriting can produce working sets that have a large number of QPRs. At every step in the rewriting process, *every* QPR in the current working set is combined with *every* QPR in the preceding operation's QPR set. If a QPR's `pattern` has multiple elements, each of these elements must be combined with *every* QPR in the preceding operation's QPR set.

This rule explosion is compounded by the fact that many IML operations generate QPR sets with multiple rules. The `add_edge`, `union_graphs`, and `replace_*` operations all produce a minimum of 2 QPRs, while the `merge_nodes` and `split_node` operations each produce a minimum of 5 QPRs. `UNION` and `OPTIONAL` statements, as well as multiple result triples for the `extract_edges` operation, can cause additional QPRs to be added to an operation's QPR set.

Many of the QPRs in the rewritten query's QPR set may be invalid (i.e., will never produce a valid result) or redundant, yet increase the cost of query evaluation. In the next two sections, we describe optimizations to both slow this rule explosion, by eliminating invalid or redundant rules, and make evaluation of the generated rules more efficient. We use "QPR" and "rule" interchangeably.

4.4 Rule-based optimizations

Our rewriting engine applies rule-based optimizations to the working QPR set after rewriting over an IML operation; we describe them in this section. Several of these optimizations use RDF characteristics to eliminate invalid rules.

Constraint Simplification The Constraint Simplification optimization identifies and eliminates those rules whose `constraints` will *always* be false. This optimization simplifies a rule by evaluating expressions in its `constraints`, focusing on equality and inequality expressions.

Bound Template Variables The Bound Template Variables optimization eliminates rules where the variables in the `template` are not used in the `pattern`; these rules will never produce a valid result. This scenario often occurs when an operation's `WhereClause` contains a `UNION` or `OPTIONAL` statement.

RDF Literal Semantics RDF requires both subjects and predicates to be URIs, not literals. The RDF Literal Semantics optimization eliminates rules that have a `pattern` or `template` with a literal in the subject or predicate position.

Literal Range Paths RDF Schema enables ranges to be associated with specific properties in an RDF graph. Thus, if a property `p` has a literal range (e.g. integer, boolean), any triple with predicate `p` will have a literal as the object. RDF restricts literals to the object position in a triple. The Literal Range Paths optimization eliminates any QPR whose `pattern` tries to match a path of length 2, when the first triple in the path has a property with a literal range. Literal range properties are provided on a per-ontology basis and only applied to `WhereClauses` evaluated against an unmodified input ontology.

Observed Paths The Observed Paths optimization relies on the fact that not all pairs of properties will be directly connected via a single resource. If a QPR pattern contains a path that is *never* observed in the underlying RDF ontology, the pattern will never match the data and the rule can be eliminated. This optimization uses a per-ontology list of property pairs to eliminate QPRs; it is applied to patterns that are matched against the unmodified RDF ontology.

Query Containment The Query Containment optimization determines which, if any, of the rules in a QPR set are redundant and eliminates them. If `QPR1` is contained by `QPR2`, then `QPR1` will contribute a subset of the triples generated by `QPR2`; `QPR1` is redundant and can be removed from the QPR set.

This optimization checks if a QPR is contained by another QPR by determining if there is a homomorphism between the two QPRs. Let `q1` and `q2` represent two QPRs, and let t_{q1} and t_{q2} represent triples that are produced by `q1` and `q2`, respectively. A homomorphism $f : q2 \rightarrow q1$ is a function $f : variables(q2) \rightarrow variables(q1) \cup constants(q1)$ such that: (1) $f(body(q2)) \subseteq body(q1)$ and (2)

$f(t_{q1}) = t_{q2}$. The homomorphism theorem states that $q1 \subseteq q2$ iff there exists a homomorphism $f : q2 \rightarrow q1$.

To determine if there is a homomorphism between two QPRs, we map their `patterns` to a SAT problem. If a possible solution is found by a SAT solver, meaning that `QPR1` is contained by `QPR2`, we substitute the values from the homomorphic mapping into both QPRs' `constraints` to determine if `QPR1` is indeed contained by `QPR2`.

The scenario in which one query is contained by another often occurs when an operation's `WhereClause` contains an `OPTIONAL` statement. The query containment optimization is only applied when rewriting over a view with this property.

4.5 Performance Optimizations

When converting the final rewritten QPR set into a vSPARQL query, we use optimizations to reduce the cost of evaluating the rewritten query.

Query-Template Collapse The Query Template Collapse optimization identifies QPRs with identical `patterns` and `constraints`, and produces a single vSPARQL subquery with multiple triples in the `CONSTRUCT` template.

Query Minimization Rewriting a query over a view can produce QPRs with redundant `pattern` elements. This often happens when multiple elements in the `WhereClause` need to be rewritten over the same IML view. The Query Minimization optimization detects and eliminates redundant elements in a `pattern`.

This optimization builds upon our Query Containment optimization. For a given QPR `q`, we remove a single `pattern` element `wc` to create `q'`. If `q'` is contained by the original query `q`, we can remove `wc` permanently from `q`; if not, `wc` is not redundant and must remain in `q`. We repeat this process for each element in the QPR's `pattern`.

Statistics-based Query Pattern Reordering IML's rewriting engine converts QPRs into vSPARQL subqueries. Each QPR's `pattern` and `constraints` are converted into a `WHERE` clause. During conversion, the rewriter has two goals, to minimize the cost of evaluating: 1) individual QPRs, and 2) all QPRs.

The rewriter uses per-ontology statistics, shown in Table 2, to achieve these two goals. These statistics are used to assign an expected triple result set size to individual elements in a QPR `pattern` based upon Table 3. Elements connected via a shared variable are grouped and then ordered with the more selective elements first. For query patterns containing property path expressions, we estimate the triple set size using the estimated fan in (fan out) of the path. The `fanInPath(pE)` function recursively calculates the fan in of a path expression.

Table 2. Per RDF Graph Statistics.

Overall Graph Statistics		**For Each Property p**	
Total Triples	Average Fan In Degree	# Total Triples	Fan Out(p)
# Distinct Subjects	Average Fan Out Degree	# Distinct Subjects	Fan In(p)
# Distinct Predicates	Average PPlus Length	# Distinct Objects	Fan Out(p+)
# Distinct Objects			Fan In(p+)

Table 3. Query pattern vs. estimated triple set size vs. estimated variable cardinality

Query Pattern	Est. Tuple Set Size	Est. Variable Cardinality
?a ?b ?c	TotalTriples	?a = #DistSubjects, ?b =#DistProperties ?c=#DistObjects
?a ?b z	AvgFanIn	?a = AvgFanIn, ?b = AvgFanIn
?a y ?c	TotalTriples(y)	?a = #DistSubjects(y), ?c = #DistObjects(y)
?a y z	FanIn(y)	?a = AvgFanIn
x ?b ?c	AvgFanOut	?b = AvgFanOut, ?c = AvgFanOut
x ?b z	min(AvgFanIn, AvgFanOut)	?b = min(AvgFanIn, AvgFanOut)
x y ?c	FanOut(y)	?c = FanOut(y)
x y z	1	
?a (pE) ?c	TotalTriples-1	?a = ?c = max(#DistSubjects, #DistObjects)
?a (pE) y	fanInPath(pE)	?a = fanInPath(pE)
x (pE) ?c	fanOutPath(pE)	?c = fanOutPath(pE)
x (pE) y	1	

```
fanInPath(pE):                    // Uses statistics in Table 2
   Property(prop): FanIn(prop)
   Inverse(Property(prop)): FanOut(prop)
   Alternate(lpath,rpath): fanInPath(lpath) + fanInPath(rpath)
   Sequence(lpath, rpath): fanInPath(lpath) * fanInPath(rpath)
   Modified(path):  if( ZeroOrMore(path) || OneOrMore(path) )
                        fanInModPath(path)

fanInModPath(pE):                        // For p+ and p* paths
   Property(prop): FanInPlus(prop)
   Inverse(Property(prop)): FanOutPlus(prop)
   Alternate(lpath, rpath): fanInPath(lpath+)+fanInPath(rpath+)
   Sequence(lpath, rpath): fanInPath(lpath+)*fanInPath(rpath+)
   Modified(path):  c = fanInPath(path.subpath)
                    if( ZeroOrMore(subPath) || OneOrMore(subPath) )
                        c *= AvgPPlusLength
```

The rewriter repeatedly chooses the most selective `WHERE` clause element, with preference for those containing already visited variables, and adds it to the query pattern. Individual `constraint` expressions are ordered after their corresponding variables have been initially bound. As elements are added to the `WHERE` clause, the rewriter updates the expected cardinality of its variables. The updated cardinalities are used to recalculate triple set sizes and choose the next `WHERE` clause element to add to the BGP. This is similar to the algorithm in [27]; we do not use specialized join statistics.

Property Path Expression Direction Using estimated variable cardinalities and the `fanInPath(pE)` routine, the rewriter estimates the number of graph nodes that will be touched by evaluating a property path expression in the forward and reverse directions. If the cost of evaluating a path expression is decreased by reversing its direction, the rewriter reverses the path before adding it to the rewritten `WHERE` clause.

Anchored Property Path Subqueries The same property path expression may appear multiple times in the same QPR or in many different QPRs. If these expressions have a shared constant as either the subject or object value, we call

them anchored property path expressions. The rewriter determines if it is more efficient to evaluate and materialize a repeated anchor property path expression once in a subquery, or to evaluate each property path expression individually.

The rewriter creates a subquery for an anchored path expression in two situations: 1) if an anchored property path expression occurs in more than three different QPRs; and 2) if a single QPR has multiple instances of the same anchored path expression and their evaluation cost is greater than three times the cost of evaluating the anchored path expression in a subquery.

5 Implementation

Our system, developed in Java, produces vSPARQL queries that can be evaluated by the execution engine – an extension to Jena's ARQ – described in [26]. The rewriting engine converts constraints to conjunctive normal form when they are added to a QPR, eliminating the need for refactoring during rewriting. MiniSAT [3] is used for solving the SAT problems generated for our Query Containment optimization. To reduce the impact of repeated property path expressions in multiple QPRs, we have added a LRU path cache to the vSPARQL query engine. The path cache stores the result of evaluating individual property path expressions, keyed on the (source URI, property path expression) pair.

The query rewriter does not rewrite all IML operations; most notably, we do not rewrite `extract_recursive` and certain property path expressions. The operation and its dependencies are converted to nested vSPARQL subqueries.

6 Evaluation

We evaluate our query rewriting system on the use case view definitions described in [26]. These view definitions transform one or more of four RDF biomedical ontologies: NCI Thesaurus [4], Reactome [7], Ontology of Physics for Biology [12], and the Foundational Model of Anatomy [23]. Although IML can express all of the transformations, the presence of `extract_recursive` operations late in two view definitions prevent beneficial query rewriting.

Table 4 presents the RDF triple size statistics for our views and queries. We use vSPARQL queries evaluated over on-demand, in-memory materialized views as our baseline query performance. These queries and views incorporate the improvements identified through query rewriting; thus rewriting performance benefits are the result of eliminating unnecessary transformations.

For this work, all view and query combinations are executed on a Intel Xeon dual quad core 2.66GHz 64-bit machine with 16GB of RAM. The computer runs a 64-bit SMP version of RedHat Enterprise Linux, kernel 2.6.18. PostgreSQL 8.3.5 is used for backend storage of the Jena SDB and is accessed using PostgreSQL's JDBC driver version 8.3-603. We use 64-bit Sun Java 1.6.0.22.

We evaluate each view and query combination five[4] times and the smallest execution time is reported. Between each run we stop the PostgreSQL server, sync the file system, clear the caches, [5] and restart the PostgreSQL server.

Table 4. Use case view and query size statistics. Entries are the number of RDF triples in the input ontology, view, or query result. Numbers in parenthesis are the number of new triples added to the base ontology by the view. Individual queries are referred to by their view and query number; for example, v2q1 is Craniofacial view query1.

	Mitotic Cell Cycle (v1)	Cranio-facial (v2)	Organ spatial location (v3)	Neuro FMA Ontology (v4)	NCI Thesaurus Simpli-fication (v5)	Bio-simulation model editor (v6)	Blood contained in heart (v7)	Radiol-ogist liver ontology (v8)	Blood fluid properties (v9)
FMA		1.67M	1.67M			1.67M	1.67M	1.67M	1.67M
FMA*				1.7M					
NCIt					3.37M				
Reactome	3.6M								
OPB									1,992
view	37	4,104	175	72,356	3.37M (180)	38	72	413	3,016 (1024)
query1	6	1	2	2	21	13	3	4	64
query2	4	4	3	10	9	1	1	9	16
query3	2	2	6	59	1	5	4	6	10
query4	5	24	2	17	6	3	2	3	41
query5	4	2	2	1		0	6	9	1
query6	5	3	1			1	13	5	
query7		592	1			3		1	
query8		520	1					1	
query9								46	
query10								1	
# subq in view def	1	3	2	17	0	4	3	7	2
time to materialize view (secs)	3.54	23.84	23.22	254.35	392.58	5.01	25.23	110.53	4.81

6.1 Rule explosion and rule optimizations

Our optimizations reduce rule explosion during rewriting, decreasing evaluation time of rewritten queries. Table 5 presents the number of rules in the QPR set generated for each view and query combination, with and without optimizations.[6] Rule optimizations are able to curb the size of QPR sets for many rewritten queries. For example, the Bound Template Variable, Query Containment, and Constraint Simplification optimizations offset the impact of `OPTIONAL` and multiple `CONSTRUCT` templates in the Organ Spatial Location's queries 5 and 6.

[4] Due to its long execution time, the Organ Spatial Location view, optimized with no path cache, is evaluated 3 times.

[5] Caches are cleared by writing "3" to /proc/sys/vm/drop_caches.

[6] The reported number of rules using optimizations is the number of vSPARQL subqueries in the generated query. N/A indicates we were unable to rewrite a query.

Table 5. Number of generated rules for each view and query (fewer is better). The first number is rules generated without optimizations. The second parenthesized number is rules generated using all optimizations. N/A indicates we could not rewrite the query.

	Mitotic Cell Cycle (v1)	Cranio-facial (v2)	Organ spatial location (v3)	Neuro FMA Ontology (v4)	NCI Thesaurus Simplification (v5)	Bio-simulation model editor (v6)	Blood contained in heart (v7)	Radiologist liver ontology (v8)	Blood fluid properties (v9)
view	3 (1)	6 (2)	4 (2)	N/A	10 (2)	N/A	4 (2)	192 (72)	5 (2)
query1	4 (4)	16 (4)	20 (4)	N/A	40 (3)	N/A	3 (3)	256 (62)	4 (1)
query2	N/A	2 (2)	16 (4)	N/A	32 (2)	N/A	3 (3)	12 (6)	20 (7)
query3	N/A	2 (2)	4 (2)	N/A	16 (1)	N/A	4 (2)	1296 (324)	N/A
query4	4 (4)	N/A	64 (2)	N/A	16 (1)	N/A	16 (14)	2208 (300)	160 (17)
query5	2 (2)	72 (16)	8192 (3)	N/A		N/A	4 (3)	N/A	4 (1)
query6	1 (1)	6 (4)	4096 (3)			N/A	12 (12)	N/A	
query7		8 (4)	64 (2)			N/A		64 (6)	
query8		8 (4)	64 (2)					60 (18)	
query9								2592 (180)	
query10								2208 (300)	

6.2 Best rewritten query performance

The time needed to materialize each use case's vSPARQL view definition is presented in Table 4. We compare times for rewriting and evaluating IML queries to the on-demand evaluation of queries over these vSPARQL views.

We evaluate each query with several different optimizations. First we rewrite each query using all of the performance optimizations described in Section 4.5 and evaluate for path cache sizes of 0MB and 4MB. Next, for 4MB path caches, we rewrite and evaluate the query without anchored property path subqueries, and we rewrite and evaluate the query without Query Minimization.

Figure 4 compares the best baseline query and rewritten query execution times. The chart displays the percentage difference from the baseline vSPARQL execution time. If a rewritten query takes the same time as the baseline query, it will have value 0 on the chart; a query that takes two times the baseline execution time will have a value of 100.

Most queries are able to benefit substantially from query rewriting. 29 of the 41 queries (71%) achieve at least a 10% improvement over the baseline execution times; 25 of the 41 queries (60%) have execution times that are 60% less than the baseline. These results indicate that rewriting can significantly improve performance for a majority of our queries.

13 queries' evaluation times do not improve by more than 10%; 10 of these queries take longer to evaluate. 8 of these queries are over the Blood Fluid Properties (v9) and Mitotic Cell Cycle (v1) views and have baseline execution times of less than 5 seconds; 3 of these queries have small improvements but 5 cannot overcome the cost of rewriting. 3 of the remaining queries' (v3q2,v8q9,v8q3) patterns do not specify URIs to limit the portions of the view that they should be applied to and must be evaluated against the entire transformed view; rewriting introduces redundancy and increases execution time. Query v7q5 introduces a concrete URI in a `FILTER` expression; our rewriter does not yet benefit from

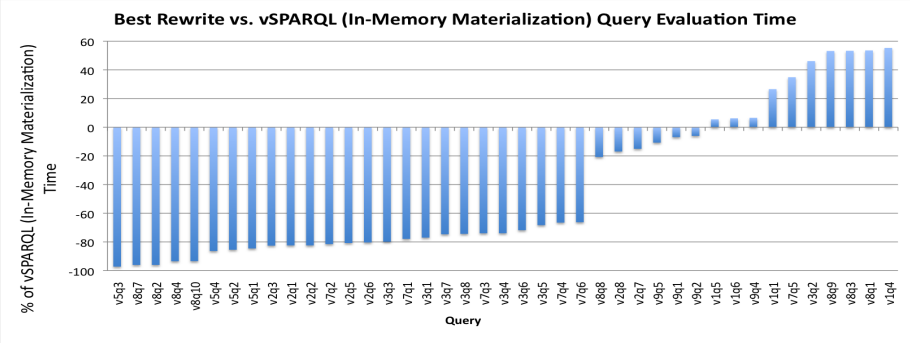

Fig. 4. Best case rewritten query vs. vSPARQL (in-memory materialization) execution time. We plot the rewritten query's execution time as the percentage difference from the baseline execution. The best query (v5q3) was 97% faster than materializing the baseline vSPARQL view in memory. Some queries (on the right) performed worse.

URIs introduced in this manner. Query v8q1 does not benefit from rewriting. The view extracts the subclass hierarchy of `fma:Organ` and replaces two direct subclasses with their (four) children; we then extract the subclass hierarchy for each of these four children, instead of once, thus increasing execution time.

6.3 Impact of rewriting options

We consider the impact of the rewriting options on our results. For space reasons, we only discuss the Organ Spatial Location view's performance, seen in Fig. 5.

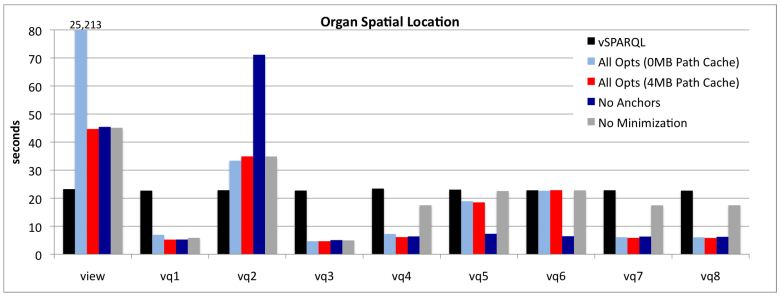

Fig. 5. Effect of rewriting options on the Organ Spatial Location view

Our Anchored Property Path optimization defines subqueries to prevent expensive property path expressions from being repeatedly evaluated. If a query does not introduce a URI not seen in the view, anchored subqueries can prevent repeated evaluation of a property path expression; this is seen with query 2. However, in the case of selective queries like query 5 and 6, rewriting generates

rules whose `WHERE` clauses do not need to be completely evaluated; the first few elements in the `WHERE` clause determine that it will never match the data, and expensive path expressions are not evaluated. For these queries, anchored paths, which are always evaluated, increase query execution time.

Query minimization can eliminate redundancy in rewritten queries. It is needed to improve evaluation of queries 4-8. These queries contain multiple `WHERE` clause elements joined by a shared variable; when the elements are rewritten over a common view, duplicate `WHERE` clause elements are created. Query minimization eliminates these duplicates and reduces execution time.

The path cache can eliminate or reduce repeated evaluation of the same property path expression. In Fig. 5, the absence of a path cache results in a large increased execution time for the rewritten view.

The rewriter provides significant benefits for queries that introduce URIs that are not in the view definition. Rewriting can improve queries by determining the most efficient direction for evaluating property path expressions. Anchored path subqueries can be used when new URIs are not introduced by the query to prevent expensive property path expressions from being repeatedly evaluated. Query minimization should be used when multiple query pattern elements are joined by a shared variable.

7 Related Work

Scripting and visual pipe transformation languages [8][17][9] allow users to specify a sequence of operations to create mashups and transform RDF. However, users must develop modules that provide the functionality available in visual editors. Evaluating queries over these transformations can be expensive; typically the entire transformed ontology is materialized and the query evaluated on it.

Visual editors are often used for transforming existing ontologies. Table 6 compares IML with the functionality provided by a visual ontology editor commonly used by bioinformatics researchers: Protege and its plug-in PROMPT. Protege is a visual editor for creating and modifying ontologies. It centers development around the subclass (i.e. "is a") hierarchy, with additional properties and values assigned to classes in this hierarchy. PROMPT provides functionality for extracting information from an ontology by traversing specified paths or combinations of paths. It also supports merging and comparing ontologies.

IML applies a sequence of transformations to a data set. These transformations can also be achieved using nested queries. Until recently SPARQL has not included support for subqueries. Schmidt [25] has developed a set of equivalences for operations in the SPARQL algebra that can be used for rewriting and optimizing queries; this work pre-dates subqueries. There has been considerable research on optimizing nested queries for relational databases[11]. Optimization of XQuery's nested FLWOR statements has focused on the introduction of a groupby operator to enable algebraic rewriting [19][22][28], including elimination of redundant navigation[13]. Most related to our work is [16], which minimizes XQuery queries with nested subexpressions whose intermediate re-

Table 6. Protege/PROMPT vs. IML Functionality

Protege & PROMPT	IML Operations
Extract edges, hierarchies	extract_edges, extract_cgraph
Delete resources, properties, values, (IS_A)hierarchies	delete_node, delete_property, delete_edge, delete_cgraph
Move resources, (IS_A) hierarchies	replace_edge_subject, replace_edge_property, replace_edge_object, replace_edge_literal
Rename resources, properties	replace_node, replace_property
Add resource, property, value	add_edge
Merge resources	merge_nodes
Combine ontologies	union_graphs
	extract_reachable, extract_path, extract_recursive
	split_node
	copy_graph

sults are queried by other subexpressions. The rewrite rules recursively prune nested queries, eliminating the production of unnecessary intermediate results, and creates a simplified, equivalent Xquery query.

The XQuery Update Framework (XQUF)[10] extends XQuery with transformation operations. Bohannon[14] presents an automaton-based technique for converting XQUF transform queries, and user queries composed with transform queries, into standard XQuery; the generated query only accesses necessary parts of the XML document. This work only addresses queries containing a single update expression. Fegaras[15] uses XML schemas to translate XQUF expressions to standard XQuery, relying on the underlying XQuery engine for optimization.

Several works have investigated rewriting SPARQL queries for efficient evaluation. Stocker[27] uses statistics on in-memory RDF data to iteratively order query pattern edges based on their minimum estimated selectivity. RDF-3X[20] optimizes execution plans using specialized histograms and frequent join paths in the data for estimating selectivity of joins. Our algorithm for producing efficient SPARQL queries is similar to an approach in[27], but adds in statistics specifically for property-path expressions.

8 Conclusions

We have presented a transforming view definition language IML for manipulating RDF ontologies. The language consists a small set of graph transformations that can be combined in a dataflow style. Our rewriting system for IML leverages the language's dataflow and compositional characteristics to rewrite queries over transforming views. We evaluated our rewriting system by defining transforming views over a set of use cases over RDF biomedical information sets.

References

1. Annoteimage. `http://sig.biostr.washington.edu/projects/AnnoteImage/`
2. Knoodl. `http://knoodl.com/`
3. The minisat page. `http://minisat.se`

4. Ncithesaurus. http://nciterms.nci.nih.gov
5. Neon toolkit. http://neon-toolkit.org/
6. The protege ontology editor and knowledge acquisition system. http://protege.stanford.edu
7. Reactome. http://www.reactome.org
8. Sparqlmotion. http://www.topquadrant.com/products/SPARQLMotion.html
9. Sparqlscript. http://www.w3.org/wiki/SPARQL/Extensions/SPARQLScript
10. Xquery update facility 1.0. http://www.w3.org/TR/xquery-update-10/
11. Chaudhuri, S.: An overview of query optimization in relational systems. In: PODS '98. pp. 34–43. ACM, New York (1998)
12. Cook, D.L., Mejino, J.L., Neal, M.L., Gennari, J.H.: Bridging biological ontologies and biosimulation: The ontology of physics for biology. In: American Medical Informatics Association Fall Symposium (2008)
13. Deutsch, A., Papakonstantinou, Y., Xu, Y.: The next framework for logical xquery optimization. In: VLDB '04. vol. 30, pp. 168–179. VLDB Endowment (2004)
14. Fan, W., Cong, G., Bohannon, P.: Querying xml with update syntax. In: SIGMOD '07. pp. 293–304. ACM, New York (2007)
15. Fegaras, L.: A schema-based translation of xquery updates. In: Intl. XML database conference on Database and XML technologies. pp. 58–72. Springer-Verlag, Heidelberg (2010)
16. Gueni, B., Abdessalem, T., Cautis, B., Waller, E.: Pruning nested xquery queries. In: Conf. on Information and knowledge management. pp. 541–550. ACM, New York (2008)
17. Le-Phuoc, D., Polleres, A., Hauswirth, M., Tummarello, G., Morbidoni, C.: Rapid prototyping of semantic mash-ups through semantic web pipes. pp. 581–590. WWW '09, ACM, New York (2009)
18. Magkanaraki, A., Tannen, V., Christophides, V., Plexousakis, D.: Viewing the semantic web through rvl lenses. Web Semantics 1(4), 359–375 (Oct 2004)
19. May, N., Helmer, S., Moerkotte, G.: Strategies for query unnesting in xml databases. ACM Trans. Database Systems 31, 968–1013 (Sept 2006)
20. Neumann, T., Weikum, G.: Rdf-3x: a risc-style engine for rdf. vol. 1, pp. 647–659. VLDB Endowment (Aug 2008)
21. Noy, N.F., Musen, M.A.: Specifying ontology views by traversal. In: International Semantic Web Conference. LNCS, vol. 3298, pp. 713–725. Springer (2004)
22. Re, C., Simeon, J., Fernandez, M.: A complete and efficient algebraic compiler for xquery. In: ICDE 2006. IEEE Computer Society (2006)
23. Rosse, C., Mejino, Jr., J.L.V.: A reference ontology for biomedical informatics: the foundational model of anatomy. Journal of Biomedical Informatics 36(6) (2003)
24. Schenk, S., Staab, S.: Networked graphs: a declarative mechanism for sparql rules, sparql views and rdf data integration on the web. In: WWW '08. pp. 585–594. ACM, New York (2008)
25. Schmidt, M., Meier, M., Lausen, G.: Foundations of sparql query optimization. In: ICDT 2010. pp. 4–33. ACM, New York (2010)
26. Shaw, M., Detwiler, L.T., Noy, N., Brinkley, J., Suciu, D.: vsparql: A view definition language for the semantic web. Journal of Biomedical Informatics 44(1) (Feb 2011)
27. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: Sparql basic graph pattern optimization using selectivity estimation. In: WWW '08. pp. 595–604. ACM, New York (2008)
28. Wang, S., Rundensteiner, E.A., Mani, M.: Optimization of nested xquery expressions with orderby clauses. Data Knowledge Eng. 60 (Feb 2007)