SUPPORTING DYNAMIC QUERIES AND ANNOTATIONS OVER DATA GRAPHS

BY

JOANA MATOS FONSECA DA TRINDADE

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Professor Marianne Winslett

ABSTRACT

When managing large-scale graph structured data, such as those derived from social networks and dynamic distributed systems, we often need to associate metadata with whole subgraphs of data. In particular, provenance and trustworthiness are examples of metadata that can be associated to entire sugbraphs. To the extent of our knowledge, however, little work has focused on the problem of representing and querying relationships between graphs. In addition, previous research has mainly focused on annotations for static data. To support recursive and dynamic provenance annotations, we propose representing metadata as dynamic annotations over dynamic data graphs. Specifically, we extend the RDF data model and SPARQL with the notion of variables and named query graphs (i.e., dynamic graphs defined by a SPARQL query) as first-class citizens. By doing so, we allow statements where subjects and objects in RDF triples are dynamic graphs, thus enabling the representation and querying of relationships between graphs. Finally, we define the semantics of an inheritance property for relationships between dynamic graphs, and we study the problems of query containment and query composition in the context of join operations between dynamic graphs.

ACKNOWLEDGMENTS

This thesis was the result of work started during a Summer internship at the IBM T. J. Watson Research Center as part of the INARC project, under the Unified Data Analytics Group. None of this would have been possible without the excellent guidance of Dr. Anastasios (Tasos) Kementsietsidis, my mentor during this project at IBM. I am truly thankful to him for giving me the opportunity to be an intern at his project, and for letting me take complete ownership of it, even though I had no prior experience in the more theoretical side of data management.

I also would not have made it here without my advisor, Professor Marianne Winslett. She believed in me without restraints, despite my lack of a 4.0 GPA during my Brazilian undergraduate degree. Since the very beginning I felt a most welcoming and positive atmosphere at our research group. She gave us freedom to choose our own projects, and I had lots of fun working on each one of them.

I also would like to thank Tom Scavo, my mentor for Google Summer of Code 2008, and Dr. Zbigniew Kalbarczyk and Professor Ravishankar Iyer for taking me into their group as a visiting scholar. Last but not least, I would like to thank all the people at UIUC who were part of my life during these last two years.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS vi

CHAPTER 1 INTRODUCTION 1
1.1 Motivation 1
1.2 Objectives 3
1.3 Contributions 3
1.4 Thesis Organization 4

CHAPTER 2 PRELIMINARIES 5
2.1 RDF 5
2.2 Named Graphs 7
2.3 SPARQL 8

CHAPTER 3 RDF AND DYNAMIC GRAPHS 10
3.1 The Need for Representing Dynamic Graphs 10
3.2 Dynamic Data and RDF Blank Nodes 11
3.3 RDF+V and Named Query Graphs 16
3.4 RDF+Q 16
3.5 SPARQL+Q 17
3.6 Concrete Syntax Examples 18
3.7 RDF+Q Statement 18
3.8 Named Query Graph 18
3.9 Properties 19
3.10 SPARQL+Q Queries 19

CHAPTER 4 IMPLEMENTATION 22
4.1 Evaluating SPARQL+Q Queries 22
4.2 Join As Query Rewriting 22
4.3 Proof-of-Concept Prototype 23
4.4 Architecture 23
4.5 Visual Query Editor 26
4.6 Query Evaluator Layer 29
4.7 Storage Layer 29
4.8 Result Converter Layer 30
4.9 Result Presentation Layer 32

CHAPTER 5 RELATED WORK 33

CHAPTER 6 CONCLUSION 35

REFERENCES 36

LIST OF ABBREVIATIONS

ANTLR	ANother Tool for Language Recognition
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
N3	Notation3
PDE	Plug-in Development Environment
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
W3C	World Wide Web Consortium

CHAPTER 1

INTRODUCTION

1.1 Motivation

When managing large-scale graph structured data, such as those derived from social networks and dynamic distributed systems, one often needs to associate metadata with entire subgraphs of data. Provenance and trustworthiness [1, 2] are examples of metadata that can be associated to entire subgraphs of data. For instance, the provenance of a single document may involve an entire graph of individuals that contributed to it. In addition, the trustworthiness of the information contained in that same document may be influenced by the flow of information within its provenance graph. Further, these types of metadata can be dynamic and change frequently.

We illustrate the need for dynamic metadata annotations with the following example drawn from existing social network services. In our scenario, data might have been extracted, mined, and clustered using algorithms such as k-means [3, 4]. Once data has been clustered, the user would like to (i) keep this information structured as a set of graphs in a database, (ii) define attributes for each graph and relationships between pairs of graphs, and (iii) perform queries over these sets of graphs. In addition, the user would like to define these graphs dynamically, and to annotate sets of graphs with graphs from different domains.

To better elucidate the requirements above, consider the example depicted in Figure 1.1. The goal here is to annotate a graph of artists in Last.FM, a social network service where users can keep track of music they listen to, with metadata from different domains. In particular, the other domains are Eventful, a database of music related events, and user mobility data extracted from cellular networks.

For clarity, we assume that all graphs in Figure 1.1 can be represented by a simple SPARQL query over an RDF version [5, 6] of the data set from which the graphs are obtained. For example, graph G_1^{lastfm} ("Trip Hop") can be represented

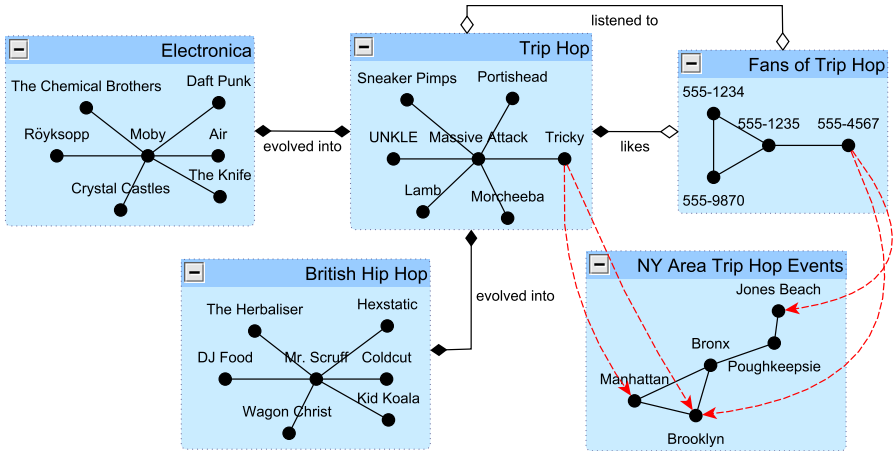

Figure 1.1: An example of graphs as annotations. Red dotted lines represent RDF predicates used for satisfying graph membership criteria (only predicates between five selected nodes are shown for clarity). Black lines between graphs represent annotation links. A filled diamond on the end of an annotation edge means that it is non-inheritant (i.e., it only applies to the graph as a unit). An empty diamond means that the annotation is inheritant (i.e., it applies to all nodes in the graph.)

by a query that returns the top-6 artists most similar to "Massive Attack", and graph G_3^{lastfm} ("British Hip Hop") by a query that returns the top-6 artists most similar to "Mr. Scruff". Similarly, graph G_2^{lastfm} ("Electronica") contains the top-6 artists in the Electronica genre. Graph $G_1^{eventful}$ ("NY Area Trip Hop Venues") contains all cities in the state of New York that have hosted at least two shows of any combination of artists in G_1^{lastfm}. Finally, graph G_1^{cdr} ("Fans of Trip Hop") contains the phone numbers of all people in a cellular network that have attended at least one Trip Hop concert from $G_1^{eventful}$ in the past 6 months.

Rather than being interested in each individual member, however, the user would like to keep data structured as graphs to make them consistent with patterns discovered through mining and clustering. Furthermore, the user intends to use these annotations to infer additional knowledge regarding musical tastes of groups of people, and to recommend to these groups artists and events of interest. Given the set of graphs described above, the user would like to annotate G_1^{lastfm} with other graphs, as shown in Figure 1.1.

Previous work has partially addressed these requirements. Carroll et al. propose in [7] an extension on the syntax and semantics of RDF graphs to enable graph naming. Specifically, their approach consists of associating an URI with a RDF graph (i.e., a set of RDF triples). This association can be used to form statements

describing RDF graphs (e.g., a triple where the subject is another RDF graph). Nevertheless, the association is a static one, in that the set of triples that compose a graph have to be defined extensionally. In our example, however, graphs are constantly changing (e.g., graph G_1^{cdr}, in which the number of users attending concerts in NY can grow periodically). Therefore, we believe that graphs should also be dynamically defined. Specifically, rather than listing all RDF triples that together form a graph, we propose to intensionally define graphs by queries that return the set of RDF triples of interest.

1.2 Objectives

Our first objective is to allow users to define RDF graphs dynamically. This way, the problem of annotating constantly changing and large-scale data can potentially be more treatable. Our second goal is to extend the RDF data model with dynamic graphs as first-class citizens, hence allowing users to annotate dynamic graphs and establish relationships between them. Finally, our last objective is to allow users to query such data and annotations visually and with an extended version of SPARQL.

1.3 Contributions

The main contributions of this thesis are summarized below:

- We are the first to treat graphs as first class citizens in the RDF data model. In addition, we define the concept of dynamic RDF graphs as those described intensionally by SPARQL conjunctive queries. We further extend the RDF data model to also treat dynamic graphs as first class citizens.
- Inspired by previous work on provenance annotations for databases [8] and RDF graphs [7], we define an inheritance property for links between dynamic RDF graphs.
- We define the semantics of a join operation between dynamic RDF graphs as SPARQL query rewriting based on previous work by Le et al. [9]).

- We implement a proof-of-concept prototype framework that supports most of the concepts defined in this thesis. As part of this prototype, we have implemented a visual query editor, a SPARQL to SQL query translator, and two different candidate SQL schemas for dynamic graphs that are used together with an existing SQL database.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 mentions the existing semantics for RDF data model and SPARQL query language. Chapter 3 describes the syntax and semantics of our extensions to RDF and SPARQL to accommodate the concept of dynamic graphs and annotations on such graphs. Chapter 4 illustrates our implementation of these concepts, including the query evaluation process for operations between dynamic graphs, and a description of our proof-of-concept prototype. Chapter 5 mentions existing related work both from research in both graph structured data, and relational databases. Finally, Chapter 6 concludes this thesis.

CHAPTER 2

PRELIMINARIES

In this chapter we introduce some of the background and theory basis used in this thesis. In particular, we introduce concepts pertaining to RDF, Named Graphs and SPARQL, briefly describe the syntax and semantics of their data models.

2.1 RDF

RDF [6], which stands for Resource Description Framework, is a W3C standard for representing graph structured data as directed relationships between named resources. Specifically, each graph is modeled as a set of statements, and a statement is formed of three elements: subject, predicate, and object. Conceptually, subjects and objects can be viewed as graph nodes, and predicates can be seen as a directed edge from the subject node to the object node. An example of RDF graph is depicted in Figure 2.1, and the accompanying N3 syntax is shown in Figure 2.2.

Although subjects and objects are most often named resources – a Literal (i.e., a primitive data type) or an URI (i.e., an URL to a resource defined in a schema) – they can also take the form of a Blank Node [11]. Blank Nodes are used in cases where a resource does not necessarily have a unique name, and yet behaves similarly to an RDF literal. Blank Nodes are also defined in SPARQL's data model, the main query language for RDF, but present a different semantics in both models. This semantic mismatch and our proposed solution for it are addressed in more details in Chapter 3.

2.1.1 RDF Reified Statements

RDF also allows metadata about triples to be described using reified statements [12]. A reified statement is a graph containing four triples (also referred to as a quad), in which a triple describes its type as an RDF statement, a triple describes

Figure 2.1: Examples of RDF graph [10]. A graph is a set of statements, and a statement is a triple containing a subject, a predicate, and an object. Predicates are depicted as depicted as directed edges, subjects as source nodes, and objects as target nodes. In this example, subjects and objects are either in Literal form (yellow boxes "Eric Miller" and "Dr.") or URI form (green ellipses).

```
me type Person .
me fullName "Eric Miller" .
me mailbox mailto:em@w3.org .
me personalTitle "Dr." .
```

Figure 2.2: N3 syntax of RDF graph example from Figure 2.1. We exclude prefixes and namespaces from the example for simplicity.

its subject, a triple describes its predicate, and the last triple describes its object. For example, consider the first triple in Figure 2.2. A reified version of it is depicted in Figure 2.3.

```
triple1 rdf:type rdf:statement .
triple1 rdf:subject me .
triple1 rdf:predicate type .
triple1 rdf:object Person .
```

Figure 2.3: The first triple in 2.1 as a RDF reified statement.

2.1.2 RDF Semantics

Having briefly introduced some of the RDF data model concepts, a more formal definition of its semantics is given below.

Let $\mathcal{U}$ be the set of all URI references, $\mathcal{B}$ the set of Blank nodes, and $\mathcal{L}$ the set of Literals as defined in [13]. $\mathcal{U}$, $\mathcal{B}$ and $\mathcal{L}$ are infinite and pairwise disjoint. An RDF statement is a tuple $(s, p, o) \in (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$, where s is the *subject*, p is the *predicate*, and o is the *object*. An RDF graph g is a set of RDF statements, and $\mathcal{G}$ is the set of all RDF graphs, i.e., the power set of $(\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$.

2.2 Named Graphs

Named Graphs is an extension to the syntax and semantics of RDF graphs proposed by Carroll et al. [7] to enable graph naming. The motivation behind it is to keep multiple graphs on the same RDF dataset, to distinguish such graphs from each other, and to establish relationships between them. Specifically, the approach consists of associating a URI with a set of triples. Two examples of Named Graphs are shown in Figure 2.4

```
:G1 { _:Monica ex:name "Monica Murphy" .
         _:Monica ex:email <mailto:monica@murphy.org> .
         _:G1 pr:disallowedUsage pr:Marketing }
:G2 { :G1 ex:author :Chris .
         :G1 ex:date "2009-09-03"^^xsd:date. }
```

Figure 2.4: Two examples of Named Graphs from [7]. Each Named Graph is defined by enclosing a set of RDF triples with brackets, and the label adjoining the set is the Named Graph's name (:G1 and :G2). This example also shows the usage of an RDF Blank Node (_:Monica) as a container for a complex type, as well as references to a Named Graph inside a Named Graph (recursive in _:G1, and to _:G1 in _:G2).

2.2.1 Named Graphs Semantics

Let $\mathcal{U}$ be the set of all URI references, and $\mathcal{GP}$ the set of Graph Patterns. As defined in [7], a Named Graph is a tuple $ng = (n, gp)$ with $n \in \mathcal{U}$, and $gp \in \mathcal{GP}$. In

addition, it has the functions $name(ng) = n$, and $graphpattern(nqg) = gp$. Also defined are the relationships "Graph", "subGraphOf" and "equivalentGraph" for pairs of Named Graphs.

2.3 SPARQL

SPARQL [14], a recursive acronym that stands for SPARQL Protocol and RDF Query Language, is the W3C query language recommendation for RDF. The anatomy of a basic SPARQL query is similar to that of SQL, with the addition of a few elements. Disregarding optional parts of the query, such as prefix declaration (i.e., namespaces associated to the RDF data to be queried), and query modifiers (e.g., ORDER BY, LIMIT, and OFFSET), a simple SPARQL query obeys the following template depicted in Figure 2.5.

```
<query_form>
FROM <dataset>
WHERE {
    <basic_graph_pattern>
}
```

Figure 2.5: Anatomy of a simple SPARQL query. A query form can be either SELECT, ASK, CONSTRUCT, or DESCRIBE. A dataset specifies one or more RDF graph datasets to be used for answering the query. Finally, a basic graph pattern is a set of RDF triples that can also contain variables as subject, predicate, and object.

To find results for a query, SPARQL relies on the concept of graph pattern matching. Just like graphs are described in RDF as a set of triples, also a SPARQL query describes a graph using a set of triples in what is called the basic graph pattern construct. In SPARQL, however, triples can contain variables as subject, predicate and object, and a variable can be matched against any element in the RDF dataset that satisfies the graph pattern constraints.

SPARQL accepts SELECT, CONSTRUCT, ASK and DESCRIBE queries. SELECT returns all or a subset of the variables bound for the basic graph pattern. CONSTRUCT returns an RDF graph constructed by matching values to the variables in the basic graph pattern. ASK returns whether the graph pattern can be matched or not, and DESCRIBE returns an RDF graph describing the resources

found in the dataset.

In this thesis, we focus on a subset of SPARQL queries. Specifically, the subset of SPARQL queries that is equivalent to conjunctive queries, i.e., SELECT queries with the basic graph pattern, and no optional graph pattern nor optional operators. It is our understanding that this subset comprises the core and most commonly used functionality of SPARQL, and still poses a challenge in terms of complexity of query answering.

CHAPTER 3

RDF AND DYNAMIC GRAPHS

In this chapter we describe the syntax and semantics of our extensions to RDF and SPARQL to accommodate the concept of dynamic graphs and annotations on such graphs. We start by showing an example of how dynamic data is currently represented in ad-hoc and error prone manner in one of the most popular and publicly available RDF datasets. We continue by describing the syntax and semantics of our solution, and how we address other open and important problems, such as a semantic mismatch between the RDF and SPARQL data models. Finally, we conclude by giving examples of query syntax using the use case described in Chapter 1.

3.1 The Need for Representing Dynamic Graphs

In this section we draw attention to the lack of support for properly representing dynamic data graphs using the current specifications of RDF. To illustrate this deficiency, we use the following example based on RDF data publicly available at DBPedia [15]. Figure 3.1 below shows one of the many ad-hoc forms that RDF data contributors use to represent dynamic data graphs (namespaces are omitted for clarity). In particular, the figure depicts an RDF snippet where the subject ("Massive Attack") is linked by a predicate ("hasPhotoCollection") to a dynamically changing collection of photos.

Furthermore, the URL http://www4.wiwiss.fu-berlin.de/flickrwrappr/photos /Massive_Attack in fact contains a web application. If a user opens this URL on her browser, it automatically executes a query on Flickr – a popular photos website – to retrieve all photos with "Massive Attack" in their description. The query results are displayed as links to actual photos, and as RDF triples generated by the application. A partial snippet of the RDF triples generated by this application is shown if Figure 3.2. This ad-hoc representation of dynamic graphs is widely

```
Massive_Attack rdf:type Band
Massive_Attack genre ‘‘Trip Hop’’
Massive_Attack currentMembers Daddy_G
Massive_Attack hometown Bristol
Massive_Attack hasPhotoCollection
 http://www4.wiwiss.fu-berlin.de/flickrwrappr/photos/Massive_Attack
```

Figure 3.1: Example of ad-hoc representation of dynamic graph data in a publicly available RDF dataset. The last triple attempts to convey dynamic data information by linking the subject "Massive_Attack" with the predicate "hasPhotoCollection" to an URL of a third-party web application that executes a query and fetches multiple data items.

deployed in DBPedia's RDF data. As an evidence of this, a Google query performed on June 16th 2011 shows thousands of results for "hasPhotoCollection" in DBPedia's domain.

Besides obvious security implications (e.g., an RDF data contributor could potentially insert an URL to a malicious website), this ad-hoc representation also poses a non-natural way of representing relationships between graphs, as well as dynamic data. It is understandable that, oftentimes, data is not available in RDF format and therefore it would not be possible to establish triples referring to those resources (each individual picture, in our example). Nevertheless, even if such data were readily available in other datasets and could easily be retrieved with a single query – as in the above DBPedia and Flickr example – there are no means for doing so using the current RDF model. Specifically, to the extent of our knowledge, there currently exist no options for storing queries or otherwise dynamic data in RDF datasets.

In this thesis, we draw from existing SPARQL and RDF specifications [14, 7] and related works in P2P and incomplete databases [16, 17] to devise our solution. Furthermore, we posit that dynamic data can be represented by extending the RDF model with entities that carry the same semantics as that of SPARQL variables.

3.2 Dynamic Data and RDF Blank Nodes

As stated previously, currently there are no means to specify dynamic data in RDF datasets. In Chapter 2, we introduced basic concepts of RDF's data model, where RDF graphs are sets of tuples called triples. Each triple is formed of a subject,

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Generated by RdfSerializer.php from RDF RAP.
# http://www.wiwiss.fu-berlin.de/suhl/bizer/rdfapi/index.html !-->

<rdf:Description rdf:about="http://dbpedia.org/resource/Massive_Attack">
   <foaf:depiction
rdf:resource="http://farm1.static.flickr.com/71/226269936_c15cb1fbf7_m.jp
g"/>
</rdf:Description>

<rdf:Description
rdf:about="http://farm1.static.flickr.com/71/226269936_c15cb1fbf7_m.jpg">
   <foaf:page
rdf:resource="http://www.flickr.com/photos/18975952@N00/226269936"/>
</rdf:Description>

<rdf:Description rdf:about="http://dbpedia.org/resource/Massive_Attack">
   <foaf:depiction
rdf:resource="http://farm1.static.flickr.com/87/258122317_22227f9960_m.jp
g"/>
</rdf:Description>

<rdf:Description
rdf:about="http://farm1.static.flickr.com/87/258122317_22227f9960_m.jpg">
   <foaf:page
rdf:resource="http://www.flickr.com/photos/45169137@N00/258122317"/>
</rdf:Description>
```

Figure 3.2: Partial snippet of the RDF results generated by the Flickr Wrapper at http://www4.wiwiss.fu-berlin.de/flickrwrappr/photos/Massive_Attack.

predicate and an object, and a predicate can only be an URI, while a subject and an object can be an URI, a Literal, or a Blank Node. Neither a Literal nor an URI can be "dynamic" in RDF's data model, i.e., once a triple has been added to the dataset, Literals and URIs cannot be "bound" to a different or more than one entity.

At first, the closest alternative for specifying free form dynamic data in RDF seems to be Blank Nodes. These are entities with a local scope that do not have a name, but rather a locally scoped label. They are often used in cases where a triple has an unknown subject, unknown object, or to establish n-ary relationships (e.g., containers for complex types). Even though Blank Nodes can be reused in different scopes, their semantics is quite similar to Literals.

For example, consider the RDF dataset in Figure 3.3, which is an extension of the dataset in Figure 3.1 with triples from a hypothetical photos RDF dataset. In addition, we also extended it with a reified statement (triple1) containing a Blank Node as subject, and with a triple with the reified statement as object. Our intention with these triples is to state that "Massive_Attack" has a photo collection (the Blank Node _:a) that consists of all photos that are tagged with "Massive Attack".

```
Massive_Attack rdf:type Band
Massive_Attack genre ‘‘Trip Hop’’
Massive_Attack currentMembers Daddy_G
Massive_Attack hometown Bristol
photo1 hasTag ‘‘Massive Attack’’
photo2 hasTag ‘‘Massive Attack’’
photo3 hasTag ‘‘Portishead’’
triple1 rdf:type rdf:statement
triple1 rdf:subject _:a
triple1 rdf:predicate hasTag
triple1 rdf:object ‘‘Massive Attack’’
Massive_Attack hasPhotoCollection triple1
```

Figure 3.3: Example of RDF dataset containing a reified statement with a Blank Node to represent dynamic data. The Blank Node is labeled ":a". The idea is to try and portray that "Massive_Attack" has a photo collection (the Blank Node _:a) that consists of all photos that are tagged with "Massive Attack". Nevertheless, it fails to do so, because the semantics of RDF Blank Nodes is similar to that of regular Literals and URIs, and Blank Nodes cannot be bound to more than one value in the same scope. As a result, the user cannot retrieve all photos that are part of the photo collection using only the reference to _:a.

In reality, however, the semantics of the statements with Blank Nodes in RDF is similar to that of regular RDF URIs and literals, and still is not sufficient to efficiently represent dynamic graphs. This is because the semantics of RDF Blank Nodes is similar to that of regular Literals and URIs, and Blank Nodes cannot be bound to more than one value in the same scope. Even though a SPARQL query that asks for all photo collections of Massive_Attack returns the Blank Node _:a, the information contained in the Blank Node triple itself does not allow for retrieving all photos in the dataset.

```
Massive_Attack rdf:type Band
Massive_Attack genre ‘‘Trip Hop’’
Massive_Attack currentMembers Daddy_G
Massive_Attack hometown Bristol
photo1 hasTag ‘‘Massive Attack’’
photo2 hasTag ‘‘Massive Attack’’
photo3 hasTag ‘‘Portishead’’
triple1 rdf:type rdf:statement
triple1 rdf:subject ?x
triple1 rdf:predicate hasTag
triple1 rdf:object ‘‘Massive Attack’’
Massive_Attack hasPhotoCollection triple1
```

Figure 3.4: Example of an extended RDF dataset containing a SPARQL variable as part of the data. The variable is labeled “x”. Unlike the blank node in Figure 3.3, the variable can be matched to any RDF subject in the dataset that satisfies it.

If, however, we replace the Blank Node in Figure 3.3 with a SPARQL variable as in Figure 3.4, then our original meaning of capturing the dynamic nature of the photo collection would be correctly portrayed. Since the semantics of a SPARQL variable is that of any matching value in the dataset, then the last triple captures precisely the entire set of RDF subjects in the dataset that have tag “Massive Attack”. In addition, any new triples added to the dataset that match against the variable would be interpreted as part of the photo collection, thus correctly capturing the dynamic nature of this data.

In the next section, we further describe how we extend RDF’s data model with SPARQL variables as first-class citizens, and how this allows for representing dynamic data.

3.2.1 Extending RDF with SPARQL variables

In this work, we posit that the semantics of RDF blank nodes alone is not enough to represent dynamic graphs. First, as it stands in the RDF data model, the current semantics does not allow dynamic graphs to be specified intensionally. Second, as pointed out by Arenas et al. [18], there is a semantic mismatch between Blank Nodes in RDF and in SPARQL. This mismatch is as follows. According to the current SPARQL's and RDF specifications ([14, 6]), Blank Nodes behave as variables when referenced in SPARQL queries, but behave as locally scoped Literals in RDF's data model.

This semantic mismatch between blank nodes in RDF and in SPARQL illustrates the need for a unified model. In this context, Arenas et al. [18] suggest the adoption of a unique semantics for both RDF and SPARQL where different Blank Nodes correspond to different entities in both the query and the data world. Unlike the authors, however, we believe that Blank Nodes are not the right solution. Furthermore, the suggestion in [18] further approximates the semantics of Blank Nodes to that of regular literals in RDF, hence obviating blank nodes as mere syntactic sugar.

Here we draw attention to the need of a unified model composed only of literals and variables that are present in both data and query worlds. This requires that variables are stored along with literals in the data, along with the definition of a semantics similar to that of incomplete databases [19]. To the extent of our knowledge, these requirements have not been addressed so far.

As such, we introduce our syntax and semantics extensions to the existing RDF data model. Specifically, we first extend the RDF data model with the concept of variables as subjects, predicates, and objects. This extension, which we name RDF+V, allows RDF statements to represent a notion of uncertainty or incomplete information, similar to previous works in relational databases [19]. Next, we extend previous work on Named Graphs [7] to support the idea of dynamic graphs (also referred to as Named Query Graphs), which are defined as Named Graphs that contain variables as part of their statements. Finally, we further extend the RDF data model to be able to establish relationships between such dynamic graphs. We accomplish this by defining RDF+Q, a data model in which the subjects and objects in statements can also be dynamic graphs.

In addition to the sets of URIs, Blank Nodes and Literals presented in Chapter 2, we incorporate the concept of SPARQL variables into the RDF data model.

A SPARQL variable can match any URI or literal, and each different match is a mapping. Two mappings μ_1 and μ_2 are said to be compatible if for every variable v in the domain(μ_1) and domain(μ_2), $\mu_1(v) = \mu_2(v)$.

3.3 RDF+V and Named Query Graphs

A RDF+V statement is a tuple $t = (u, s, p, o)$ where $u \in \mathcal{U}$, $s \in (\mathcal{U} \cup \mathcal{B} \cup \mathcal{V})$, $p \in (\mathcal{U} \cup \mathcal{V})$, and $o \in (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{V})$. That is, a reified RDF statement where the *subject*, the *predicate* and the *object* can also be a variable. We define the function $name(t) = u$. An RDF+V graph is a set of RDF+V statements, and $\mathcal{GP}$ is the set of RDF+V graphs.

A Named Query Graph is a tuple $nqg = (n, gp)$ with $n \in \mathcal{U}$, and $gp \in \mathcal{GP}$. In addition to $name(nqg) = n$, as defined in [7], we extend the definition of Named Graphs with gp as a set of RDF+V statements, and with the function $graphpattern(nqg) = gp$.

3.4 RDF+Q

Let $\mathcal{T}$ be the set of terms, such that $\mathcal{T} = (\mathcal{U} \cup \mathcal{B} \cup \mathcal{V})$, and $\mathcal{QG}$ be the set of Named Query Graphs. We define an RDF+Q statement as a tuple (n, s, p, o) where $n \in \mathcal{U}$, $s \in (\mathcal{T} \cup \mathcal{QG})$, $p \in (\mathcal{U} \cup \mathcal{V})$ and $o \in (\mathcal{T} \cup \mathcal{L} \cup \mathcal{QG})$. That is, an RDF+Q statement is an RDF+V triple where the *subject* and the *object* can also be a Named Query Graph. An RDF+Q graph is a set of RDF+Q statements.

3.4.1 Inheritance

In a RDF+Q tuple (n, s, p, o), a *predicate* p presents the inheritance property for s iff $s \in \mathcal{QG}$, and p is defined for all subgraphs of s that contain at least one node. Otherwise, p is said to be non-inheritant for s iff $s \in \mathcal{QG}$, and p is only defined for s (i.e., s as a single graph). Analogously, p presents the inheritance property for o iff $o \in \mathcal{QG}$, and p is defined for all subgraphs of o that contain at least one node. Otherwise, p is said to be non-inheritant for o iff $o \in \mathcal{QG}$, and p is only defined for o.

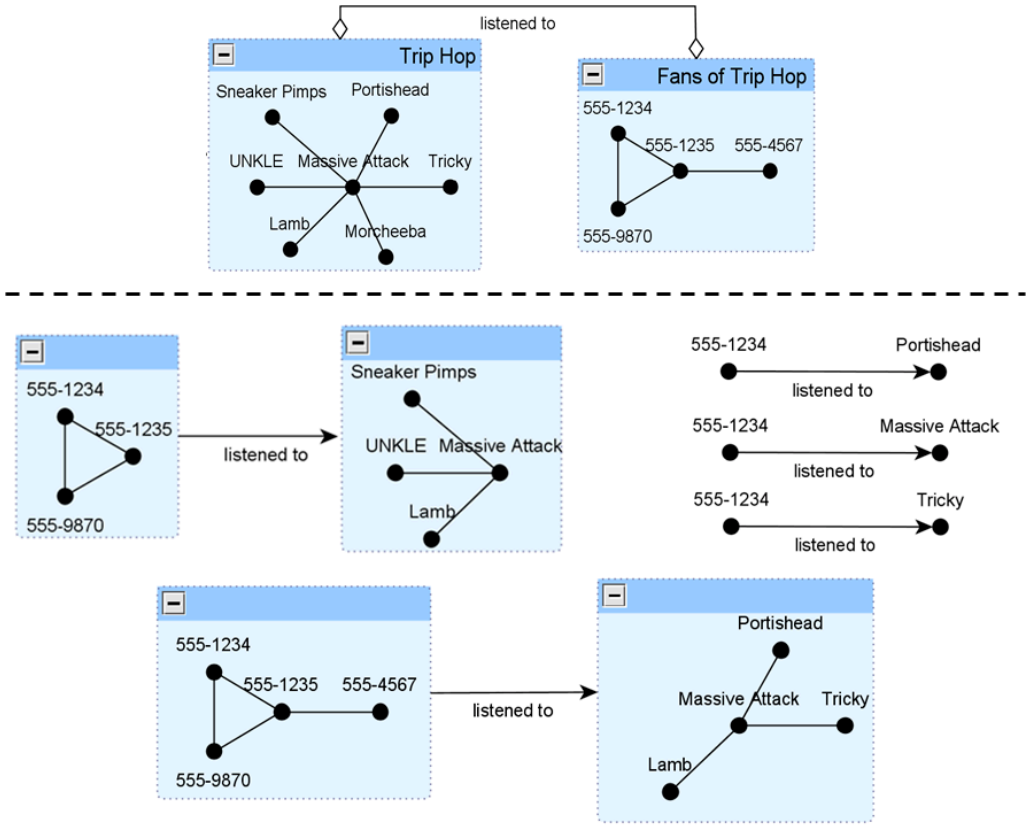

Figure 3.5: An example of inheriting predicate. The predicate is called "listened to". It has as subject the Named Query Graph "Fans of Trip Hop", as object the Named Query Graph "Trip Hop", and it is inheriting both ways. As such, the predicate is applicable to all subgraphs of both Named Query Graphs. The Figure depicts three possible interpretations of the predicate.

In other words, whenever inheritance is present in a predicate, this predicate is valid for all subgraphs of subject and/or object Named Query Graphs. A predicate between two Named Query Graphs can also be non-inheriting, resulting in graph treated as an "unit". Figure 3.5 depicts possible interpretations for an inheriting predicate between two graphs from the use case in Chapter 1.

3.5 SPARQL+Q

A SPARQL+Q query is a SPARQL query in the form `Q := (SELECT | CONSTRUCT) RD (WHERE GP)`, in which the *subjects* and *objects* of triples in GP are matched against a RDF+Q graph.

3.6 Concrete Syntax Examples

In this section we introduce examples of concrete syntax for RDF+Q statements, Named Query Graphs, inheritance property, and SPARQL+Q queries. In all examples below, the namespace rdfq refers to RDF+Q. For simplicity, reified triples are represented as quadruples where the subject follows the pattern <rdfq:statement#tripleid>.

3.7 RDF+Q Statement

Figure 3.6 depicts sample RDF+Q statements used for describing graphs G_1^{lastfm} and G_1^{cdr} from Figure 1.1, which depicts the example use case in Chapter 1.

```
<rdfq:statement#lastfm1> ?s <lastfm:similarTo>
    <$lastfm:artist#MassiveAttack> .
<rdfq:statement#cdr1> ?s <cdr:wasPresentAt> _:aCity .
<rdfq:statement#cdr2> <geo:state#NY> <geo:hasPart>
    _:aCity .
```

Figure 3.6: Sample RDF+Q quadruples.

3.8 Named Query Graph

A Named Query Graph is an augmented RDF graph where statements can also contain variables as subjects, predicates or objects. Specifically, a gp is a set of RDF+V statements combined by their names (i.e., $name(t) = u$) into a single graph using a <rdfq:containsQuad> RDF predicate. Figures 3.7 and 3.8 below depict graphs G_1^{lastfm} and G_1^{cdr} (Figure 1.1), respectively, as Named Query Graphs.

```
<rdfq:graph#TripHop> <rdfq:containsQuad$>
    <rdfq:statement#lastfm1> .
```

Figure 3.7: Graph G_1^{lastfm} ("Trip Hop") represented in RDF+Q.

```
<rdfq:graph#FansOfTripHop> <rdfq:containsQuad>
    <rdfq:statement#cdr1> .
<rdfq:graph#FansOfTripHop> <rdfq:containsQuad>
    <rdfq:statement#cdr2> .
```

Figure 3.8: Graph G_1^{cdr} ("Fans of Trip Hop") represented in RDF+Q.

3.9 Properties

Figure 3.9 depicts the concrete syntax for RDF+Q predicate properties.

```
-- Non-Inheritant Both-ways
<rdfq:predicate#evolvedInto> <rdfq:property#from>
<rdfq:property#nonInheritant> .
<rdfq:predicate#evolvedInto> <rdfq:property#to>
<rdfq:property#nonInheritant> .

-- Inheritant Both-ways
<rdfq:predicate#listenedTo> <rdfq:property#from>
<rdfq:property#inheritant> .
<rdfq:predicate#listenedTo> <rdfq:property#to>
<rdfq:property#inheritant> .

-- Inheritant and Non-Inheritant hybrid
<rdfq:predicate#likes> <rdfq:property#from>
<rdfq:property#inheritant> .
<rdfq:predicate#likes> <rdfq:property#to>
<rdfq:property#nonInheritant> .
```

Figure 3.9: Different types of RDF+Q predicates.

3.10 SPARQL+Q Queries

Figure 3.10 depicts five sample SPARQL+Q queries. The first query searches for all pairs of graphs where the first graph "listened to" the second graph. The second query searches for all pairs of graphs where the first graph has "evolved into" "Trip Hop". The third query checks if the graph formed by all nodes that share a "friend" relationship with "555-1234" "likes" the graph "Trip Hop". The fourth query searches for the graph of all users that have "listened" to both "British Trip

Hop" and "Electronica" graphs. Finally, the fifth query looks for all graphs that have "evolved into" at least one genre that users in the graph defined by "friend of 555-1234" have "listened to".

As can be seen from this example, statements in graph patterns of a SPARQL+Q query can reference two different types of entities. The first option is to refer to a specific Named Query Graph as the subject or object in a statement (e.g., second, third, and fourth queries in Figure 3.10). The second option is to use the $?g\{\}$ construct to specify new graph patterns to be matched.

```
# All graphs that have "listened to" any other graph
SELECT *
WHERE {
    ?g1 {?s1 ?p1 ?o1} <rdfq:predicate#listenedTo>
        ?g2 {?s2 ?p2 ?o2} .
}

# All graphs that have "evolved into" Trip Hop graph
SELECT *
WHERE {
    ?g {?s ?p ?o} <rdfq:predicate#evolvedInto>
        <rdfq:graph#TripHop> .
}

# All subgraphs of "friends" of user "555-1234" that
# "like" Trip-Hop
SELECT *
WHERE {
    ?g {?s <cdr:friend> <cdr1:user#555-1234>}
        <rdfq:predicate#likes> <rdfq:graph#TripHop> .
}

# The graph of users that have listened to both
# "British Trip Hop" and "Electronica"
SELECT ?g1
WHERE {
    ?g1 {?s1 ?p1 ?o1} <rdfq:predicate#listenedTo>
        <rdfq:graph#Electronica> .
    ?g1 {?s1 ?p1 ?o1} <rdfq:predicate#listenedTo>
        <rdfq:graph#BritishTripHop> .
}

# All graphs that have "evolved into" at least one
# "genre" that users in the graph defined by friends
#  of "555-1234" have listened to
SELECT ?g3
WHERE {
    ?g1 {?s <cdr:friend> <cdr1:user#555-1234>}
        <rdfq:predicate#listenedTo> ?g2 {?s2 ?p2 ?o2} .
    ?g3 {?s3 ?p3 ?o3} <rdfq:predicate#evolvedInto>
        ?g2 {?s2 ?p2 ?o2} .
}
```

Figure 3.10: Sample SPARQL+Q queries.

CHAPTER 4

IMPLEMENTATION

This chapter describes the implementation aspects of this thesis. We start by showing algorithms to evaluate SPARQL+Q queries, and to perform JOIN operations between dynamic graphs as SPARQL query rewriting. We finish by illustrating details of a proof-of-concept prototype that implements some of the ideas throughout this thesis.

4.1 Evaluating SPARQL+Q Queries

In this section we explain the overall steps to evaluate SPARQL+Q queries. The algorithms herein are an extension on previous work by Le et al. on SPARQL query rewriting [9].

4.2 Join As Query Rewriting

The steps of SPARQL+Q query evaluation consist of:

1. Given a graph pattern where each triple contains a dynamic graph as either subjects or object, we generate all possible re-orderings as a set of new graph patterns using Algorithm 4. In particular, each new graph pattern is a rewriting of the original graph pattern where a different set of dynamic graphs are joined. Additionally, we consider only the set of rewritings that are shaped as a tree (i.e., no disjoint graph pattern triples). If no reordering is possible, skip to step 3.

2. The join between two query graphs is defined as a query rewriting of one graph in terms of the other, according to algorithms 1, 2, and 3. If two graphs are joinable, then algorithm 2 produces a non-empty set of rewritings.

3. Next, we check both ends of the predicate linking two dynamic graphs. If the predicate is non-inheritant, the evaluation can only return a non-empty result if the query refers to a graph as a unit. If the predicate presents the inheritance property, then we bound the size of the minimum graph to be returned as the greatest common subgraph between the two dynamic graphs.

4. Finally, we return the set of mappings that result from evaluating the graph pattern over the default RDF dataset.

To compute the result of a JOIN between two graphs, we devise the following query rewriting algorithm based on previous work by Le et al. [9]:

- First, we consider only CONSTRUCT queries, to preserve the structure of resulting data graphs.
- We extend Le et al. SPARQL query rewriting algorithm to deal with predicates as variables. Other constants act as anchors.

Algorithm 1 Map

Require: $x, y \in \mathcal{U} \cup \mathcal{V}$, x from query, y from view
Ensure: mapping between y and x or $undefined$

Set $\Phi(y)$ to $undefined$
if $(x \in \mathcal{V} \wedge y \in \mathcal{V}) \vee (x \in \mathcal{U} \wedge y \in \mathcal{V})$ **then**
    $\Phi(y) = x$
end if
if $x \in \mathcal{V} \wedge y \in \mathcal{U}$ **then**
    $\Phi(y) = y$
end if
if $(x \in \mathcal{U} \wedge y \in \mathcal{U}) \wedge (x = y)$ **then**
    $\Phi(y) = y$
end if
Return $\Phi(y)$

4.3 Proof-of-Concept Prototype

4.4 Architecture

In this section, we describe the overall architecture of our proof-of-concept prototype and its components.

Algorithm 2 Extended SQR (ESQR)

Require: view V, query $q(\bar{X})$:– $(?S^x_1, ?P^x_1, ?O^x_1), \ldots, (?S^x_n, ?P^x_n, ?O^x_n)$
Ensure: a rewriting Q' as a union of conjunctive queries

for *each* $(?S^x_i, ?P^x_i, ?O^x_i), 1 \leq i \leq n$ **do**
  Let $HD(V)$:– $(?S^y_1, ?P^y_1, ?O^y_1), \ldots, (?S^y_m, ?P^y_m, ?O^y_m)$
  for $k = 1, \ldots, m$ **do**
    Set variable mapping Φ_{ik} to $undefined$
    for the pair (x, y) in $(?S^x_i, ?S^y_k), (?P^x_i, ?P^y_k), (?O^x_i, ?O^y_k)$ **do**
      $\Phi(y) = Map(x, y)$
      if Φ_{ik} is defined **then**
        For any variable v' in V and not in $(?S^y_k, ?P^y_k, ?O^y_k)$, Φ_{ik} maps v' to a fresh variable
        Add (V, Φ_{ik}) to $CandV$
      end if
    end for
  end for
end for
Set the query rewriting result Q' to $\emptyset$
if $\Phi_{1k_1}, \Phi_{2k_2}, \ldots, \Phi_{nk_n}$ **then**
  $HD(q' = HD(Q))$
  $BD(q') = BD(\Phi_{1k_1}(V_1), \ldots, \Phi_{nk_n}(V_n))$
  $Q' = Q' \cup q'$
end if

Algorithm 3 QGJOIN: Join between two query graphs as an application of ESQR

Require: pair of named query graphs (G_x, G_y)
Ensure: set of rewritings Q_U

Set Q_U to $\emptyset$
Set Q_1 to $\emptyset$
$Q_1 = ESQR(graphpattern(G_x), graphpattern(G_y))$
$Q_U = Q_U \cup Q_1$
Return Q_U

Algorithm 4 Graph Pattern Re-ordering

Require: set of disjoint SPARQL+Q triples G_U
Ensure: set of new SPARQL+Q triples $TREES$ as joined re-orderings (trees) of G_U

Let $TREES = \emptyset$
Let $AVAILTRIPLES = G_U$
Select triple t from $AVAILTRIPLES$
$AVAILTRIPLES = AVAILTRIPLES - t$
$TREES = TREES \bigcup t$
while ($AVAILTRIPLES != \emptyset$) **do**
    select triple t_2 from $AVAILTRIPLES$
    $AVAILTRIPLES = AVAILTRIPLES - t_2$
    Let $tempTREES = TREES$
    for each tree $t \in TREES$ **do**
        endpointsTree = all subjects and objects of t
        endpointsT2 = all subjects and objects of t_2
        for each $(x, y) \in$ Carthesian Product of endpointsTree X endpointsT2 **do**
            form new tree by inserting edge t2 in tree and merging nodes x and y (name it xy)
            tempTREES = tempTREES $\bigcup$ newTree
        end for
    end for
    $TREES = TREES \bigcup$ tempTREES
end while
Return $TREES$

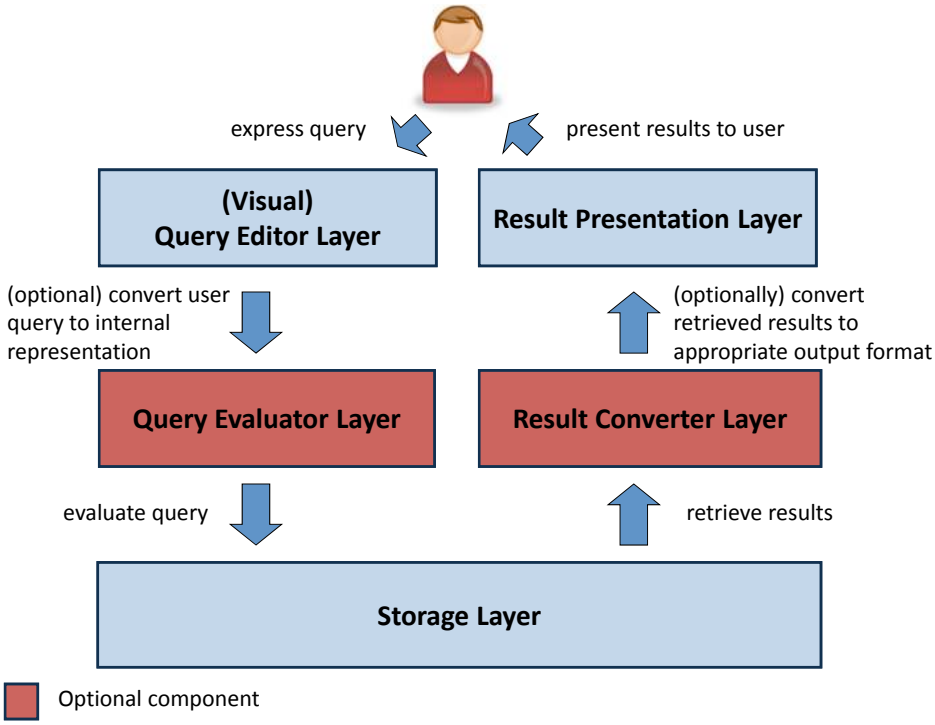

Figure 4.1: Architecture of the proof-of-concept prototype.

4.5 Visual Query Editor

The Visual Query Editor is the main user interface of the prototype. As it stands now, it is used only for building queries. The same interface might optionally be used to present the results in the future. This initial version of the user interface was built-in using Eclipse's Modeling Framework (EMF), Graphical Modeling Framework (GMF), and Plugin Development Environment (PDE). The initial EMF ecore model used as input for GMF is displayed in Figure 4.2, and a screenshot of the Visual Query Editor's current version depicted in Figure 4.3.

Currently, query graphs are represented as containers, and associated to triples by having subjects and objects contained within the query graph, as shown in Figure 4.3. Dynamic query graphs can also be collapsed or expanded to show inner triples. In addition, dynamic query graph triples will potentially have an expand capability, where results at the data level would be displayed, i.e., all the data satisfying the dynamic graph or a sample.

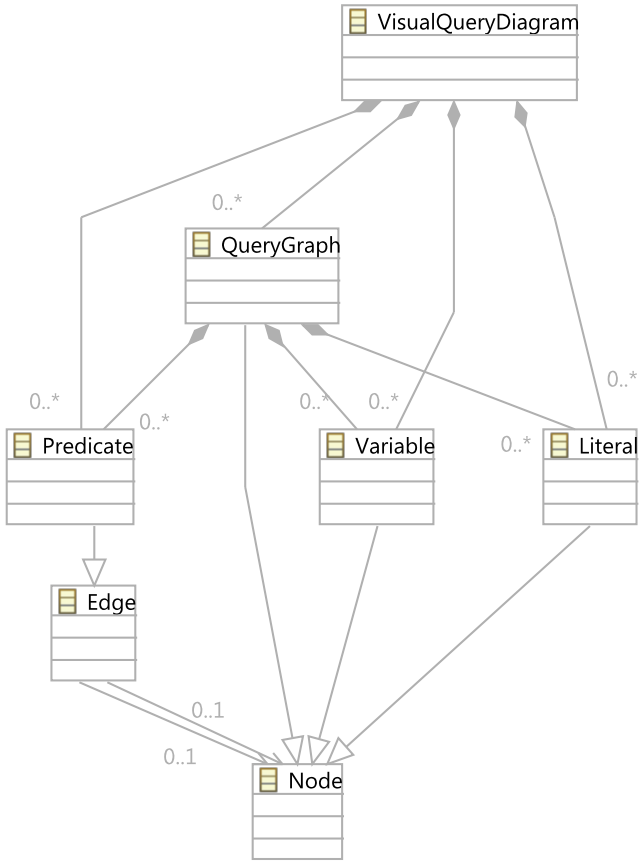

Figure 4.2: EMF Ecore model for the Visual Query Editor. This model is used as input for GMF, which in turn generates code supporting diagram creation and editing functions of the Visual Query Editor.

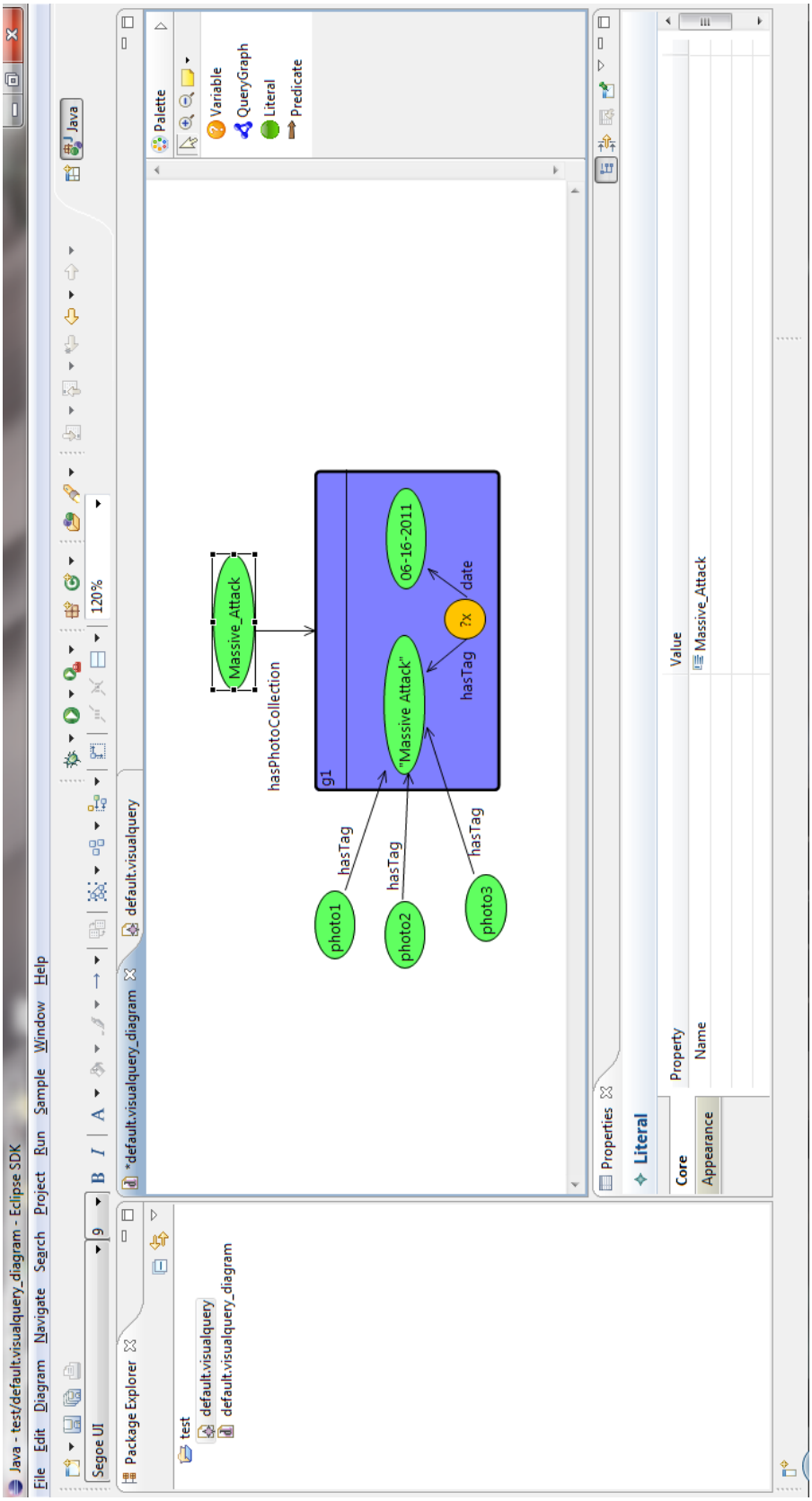

Figure 4.3: Visual Query Editor application.

4.6 Query Evaluator Layer

The Query Evaluator Layer contains the SPARQL+Q query generator, and query evaluator. Currently, a SPARQL+Q query is generated directly from the Visual Query Editor by right-clicking the visual query EMF model file, and selecting the option "Dynamic Query → Export as SPARQL+Q query". A corresponding SQL query is also currently generated in a similar fashion.

Initially, we planned on using a semantics preserving SQL to SPARQL+Q translation that, given as input a SPARQL+Q query, would generate a corresponding SQL query. To this end, we partially extended a publicly available ANTLR SPARQL grammar to generate Java classes for a SPARQL+Q parser. This parser would read a SPARQL+Q, build an AST tree, and based on a set of ANTLR tree walker rules, incrementally build a corresponding SQL query.

This plan was abandoned, however, after we realized that it was possible to incrementally generate a SPARQL+Q query and its corresponding SQL query in a much more straightforward manner directly from the XML/EMF representation of the visual query model. The XML form of the visual query diagram in Figure 4.3 generated by the Visual Query Editor is depicted in Figure 4.4.

4.7 Storage Layer

The storage layer is a SQL (relational) database, and it takes as input a SQL query equivalent of the original SPARQL+Q query. It contains representations of the RDF+Q triples expressed in RDF. This translation of RDF+QV to RDF follows the syntax of special triples presented in Chapter 3 for defining named query graphs, special subject/object values to represent variables, and special properties of predicates to represent the (non)inheritance of property values between named graphs.

In addition, the Storage Layer was designed using the PostgreSQL database management system. Two different choices of relational schemas were implemented, and these are depicted in Figures 4.5 and 4.6. In both schemas, RDF reification is emulated by using the relational tuples' id field.

The first schema, shown in Figure 4.5 consists of a single relation holding all triples, including triples with variables and triples describing dynamic query graphs. The second schema, is depicted in Figure 4.6, uses two different relations.

```
<?xml version="1.0" encoding="UTF-8"?>
<visualquery:VisualQueryDiagram xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:visualquery="http://visualquery">
  <predicates name="hasTag" source="//@literals.0"
    target="//@literals.1"/>
  <predicates name="hasTag" source="//@literals.2"
    target="//@literals.1"/>
  <predicates name="hasTag" source="//@literals.3"
    target="//@literals.1"/>
  <predicates name="hasTag" source="//@variables.0"
    target="//@literals.1"/>
  <predicates name="hasPhotoCollection" source="//@literals.4"
    target="//@querygraphs.0"/>
  <predicates name="date" source="//@variables.0"
    target="//@literals.5"/>
  <predicates name="containsQuad" source="//@querygraphs.0"
    target="//@variables.0"/>
  <variables name="?x"/>
  <querygraphs name="g1"/>
  <literals name="photo1"/>
  <literals name="&quot;Massive Attack&quot;"/>
  <literals name="photo2"/>
  <literals name="photo3"/>
  <literals name="MassiveAttack"/>
  <literals name="06-16-2011"/>
</visualquery:VisualQueryDiagram>
```

Figure 4.4: XML of example visual query shown in Figure 4.3.

One relation holds all "first layer" triples, including triples with variables, and another relation holds all triples describing dynamic query graphs.

4.8 Result Converter Layer

At the time of writing, this layer has not been fully implemented, and its functions are performed manually. Nevertheless, in this section we describe its intended automated functionality. This layer performs an operation that is opposite from the one performed by the Query evaluation layer. That is, the layer is responsible for retrieving the "raw" data from the relational storage layer that satisfy the query sent to it, and then converting those to an appropriate data instance of the RDF+Q

id	subject	predicate	object
1	a1	b1	c1
2	a1	b2	__varX
3	__g1	includes	1
4	__g1	Includes	2

Figure 4.5: Example of RDF+Q dataset in one of the implemented relational schemas. This schema consists of a single relation holding all triples, including those containing variables (__varX) and describing dynamic graphs (__g1 and __g2). RDF reification is emulated by using the relational tuples' id field.

Triples

id	subj	pred	obj
#1	a1	b1	c1
#2	a1	b2	__varX

Graphs

id	graphName	tripleId (FK)
#3	__g1	Triples.#1
#4	__g1	Triples.#2

Figure 4.6: An example of RDF+Q dataset in one of the implemented relational schemas. This schema consists of two relations. One relation holds all "first layer" triples, including triples with variables (__varX), and another relation holds all triples describing dynamic query (__g1 and __g2). RDF reification is emulated by using the relational tuples' id field.

model. The instance is such that it satisfies the input SPARQL+Q query by the user.

4.9 Result Presentation Layer

As mentioned previously, the Result Converter Layer in its current state is not fully automated. Because the Result Presentation Layer depends on its functionality, at the time of writing it is at a partial state. In this section, we describe its intended functionality.

The result presentation layer might get as input the RDF+Q data generated by the Result Converter Layer, or it might accept as input an equivalent XML representation of the data that are to be displayed graphically, as text, or both, to the viewer. The result presentation layer might be a separate module from the input query layer, or they might be the same module.

CHAPTER 5

RELATED WORK

In [8], Geerts et al. propose an approach to capture metadata as annotations over a set of tuple values, deemed "color blocks", of a relational database. In addition, the authors define a relational algebra to perform queries over color blocks. The color query language is both minimal and complete, and allows fine-grained queries (i.e., involving tuple values) on both data and annotations. We generalize the idea of color annotations by allowing metadata to be represented as entire graphs. In our model, color annotations can be thought of as an instance of graph annotations, where each graph contains a single node.

In [7], Carroll et al. propose an extension of the syntax and semantics of RDF graphs to enable graph naming. Specifically, their approach consists of associating a URI with an RDF graph (i.e., a set of RDF triples). This association can be used to form statements describing RDF graphs (e.g., a triple where the subject is another RDF graph). In addition, the authors define a vocabulary that can be used to describe graphs and relationships amongst graphs, such as "Graph", "subGraphOf" and "equivalentGraph". As a use case, Carroll et al. show the usage of graph naming in semantic web publishing and trust evaluation by demonstrating how RDF graphs can be associated to a digital signature. This approach is a straightforward extension of RDFS, and has been used in subsequent work in provenance for RDF graphs [20, 21].

Another work that treats metadata as first class citizens is proposed by Srivastava and Velegrakis in [22]. The authors achieve this by storing metadata as queries, and by treating queries as data values. In their model, a query expresses the relationship between a metadata tuple and the data that it annotates. Furthermore, the authors extend the join operation to allow joining a tuple with relations specified by queries. To check if a tuple value is annotated with a certain query, the idea is to first evaluate that query, and then to test the tuple for membership in the relation that the query describes. One of the problems with this approach, however, is that it lacks expressiveness. For example, the authors only define the

semantics of the join operation, and leave out other relational operators. In addition, mappings between data and metadata are only possible between a tuple value and a query. As a result, to create a mapping between two relations, one has to annotate each member of the source relation (i.e., all attribute values for all tuples) with a query that describes the image relation. Our approach overcomes these drawbacks by allowing links between any two dynamic graphs to be represented using a single RDF predicate.

CHAPTER 6

CONCLUSION

In this thesis we have investigated the problem of representing and annotating dynamic data in RDF datasets. To this end, we have defined a number of extensions to RDF's data model and SPARQL query language to accommodate for, what we call, dynamic graphs or Named Query Graphs. Specifically, we have made the following contributions to the state of the art in data management.

Variables as first class citizens in RDF: We have proposed extending RDF with the notion of SPARQL variables, in a way much similar to existing work in P2P and incomplete databases. This enables RDF graphs to be specified intensionally, thus naturally matching evolving dynamic data in large-scale graphs. Based on this extension, we have defined the concept of dynamic graphs. We have also extended the RDF data model with dynamic graphs as first class citizens.

Join of Dynamic Graphs as query rewriting: We have defined the semantics of joins between dynamic graphs as an extension of previous work by Le et al. [9] in SPARQL query rewriting.

Proof-of-Concept prototype: We have partially implemented the ideas presented in this thesis in a proof-of-concept prototype framework. The framework includes an initial version of a SPARQL+Q visual query editor, a SPARQL+Q query generator with support to dynamic graphs, and an underlying relational storage layer. Two different SQL schemas for representing named query graphs were proposed and implemented for the PostGreSQL database management system.

As future work, we leave further improvements in the proof-of-concept prototype, as well as a more thorough evaluation of the different alternative schemas, including but not limited to: (i) more complex queries, (ii) an analysis of the relational tables' growth and query execution speed if RDF reification is not emulated using the relational tuple's id field, and (iii) an evaluation of the performance of the proof-of-concept prototype under different settings.

REFERENCES

[1] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *SIGMOD Rec.*, vol. 34, pp. 31–36, September 2005.

[2] H.-S. Lim, Y.-S. Moon, and E. Bertino, "Provenance-based trustworthiness assessment in sensor networks," in *Proceedings of the Seventh International Workshop on Data Management for Sensor Networks*, ser. DMSN '10. New York, NY, USA: ACM, 2010, pp. 2–7.

[3] M. R. Anderberg, *Cluster Analysis for Applications*, ser. Monographs and Textbooks on Probability and Mathematical Statistics. New York: Academic Press, Inc., 1973.

[4] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, L. M. L. Cam and J. Neyman, Eds., vol. 1, Berkeley, CA. University of California Press, 1967, pp. 281–297.

[5] "Triplify," http://triplify.org, last visited in June 16th 2011.

[6] "W3C Resource Description Framework (RDF) Specification," http://www.w3.org/RDF/, last visited in June 16th 2011.

[7] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler, "Named graphs, provenance and trust," in *WWW '05: Proceedings of the 14th international conference on World Wide Web*. New York, NY, USA: ACM, 2005, pp. 613–622.

[8] F. Geerts, A. Kementsietsidis, and D. Milano, "Mondrian: Annotating and querying databases through colors and blocks," in *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, p. 82.

[9] W. Le, S. Duan, A. Kementsietsidis, F. Li, and M. Wang, "Rewriting queries on sparql views," in *Proceedings of the 20th international conference on World wide web*, ser. WWW '11. New York, NY, USA: ACM, 2011, pp. 655–664.

[10] "W3C RDF Primer," http://www.w3.org/TR/2004/REC-rdf-primer-20040210/, last visited in June 16th 2011.

[11] "W3C Resource Description Framework: Concepts and Abstract Syntax - Blank Nodes)," http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/#section-blank-nodes, last visited in June 16th 2011.

[12] "W3C RDF Semantics - reification," http://www.w3.org/TR/rdf-mt/#Reif, last visited in June 16th 2011.

[13] G. Klyne and J. J. Carrol, "Resource description framework (rdf): Concepts and abstract syntax," W3C, W3C Recommendation - 10 Feb 2004, 2004, available at http://www.w3.org/TR/rdf-concepts/.

[14] "W3C SPARQL Query Language for RDF Specification," http://www.w3.org/TR/rdf-sparql-query/, last visited in June 16th 2011.

[15] "DBPedia.org - About Massive Attack," http://dbpedia.org/page/Massive_Attack, last visited in June 16th 2011.

[16] A. Kementsietsidis, "Data sharing and querying for peer-to-peer data management systems," in *Current Trends in Database Technology - EDBT 2004 Workshops*, ser. Lecture Notes in Computer Science, W. Lindner, M. Mesiti, C. Trker, Y. Tzitzikas, and A. Vakali, Eds. Springer Berlin / Heidelberg, 2005, vol. 3268, pp. 378–385.

[17] T. Imielinski and W. Lipski, Jr., "On representing incomplete information in a relational data base," in *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, ser. VLDB '1981. VLDB Endowment, 1981. [Online]. Available: http://portal.acm.org/citation.cfm?id=1286831.1286869 pp. 388–397.

[18] M. Arenas, M. Consens, and A. Mallea, "Revisiting blank nodes in rdf to avoid the semantic mismatch with sparql," RDF Next Step Workshop - 26 Jun 2010, 2010, available at http://www.w3.org/2009/12/rdf-ws/papers/ws23.

[19] T. Imielinski and W. Lipski, Jr., "On representing incomplete information in a relational data base," in *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*. VLDB Endowment, 1981, pp. 388–397.

[20] G. Flouris, I. Fundulaki, P. Pediaditis, Y. Theoharis, and V. Christophides, "Coloring rdf triples to capture provenance," in *ISWC '09: Proceedings of the 8th International Semantic Web Conference*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 196–212.

[21] O. Hartig, "Querying trust in rdf data with tsparql," in *ESWC 2009 Heraklion: Proceedings of the 6th European Semantic Web Conference on The Semantic Web*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 5–20.

[22] D. Srivastava and Y. Velegrakis, "Intensional associations between data and metadata," in *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2007, pp. 401–412.