JPA Criteria Queries over RDF Data

Claus Stadler[1] and Jens Lehmann[2]

[1] Computer Science Institute, University of Leipzig
cstadler@informatik.uni-leipzig.de
[2] Computer Science Institute III, University of Bonn & Fraunhofer IAIS
jens.lehmann@cs.uni-bonn.de, jens.lehmann@iais.fraunhofer.de

Abstract. We present the design and implementation of a prototype system for querying RDF data via the Java Persistence API (JPA) criteria query feature. The JPA is a specification for management of (primarily, but not limited to) relational data and provides a framework for uniform storage and retrieval of Java objects using various backends. The framework provides the *Criteria API*, which enables building queries programmatically against a Java domain model and executing them on any supported backend. In this short paper, we describe our work towards supporting the Web of Data as a new backend. Our contributions comprise (i) a system design for enabling JPA compliant object/RDF mappings together with de-/serialization of object graphs as RDF, (ii) an approach for rewriting criteria queries to SPARQL queries, and (iii) a prototype implementation.

Keywords: RDF, SPARQL, JPA, Criteria Query, Query Rewriting

1 Introduction

A widely adopted practice in object oriented programming is to devise a domain model together with a data access abstraction for storing and retrieving *persistent domain objects*, referred to as *entities*. A main task of this abstraction is to facilitate the mapping between entities and the model supported by the backend. Querying and storing RDF data with object oriented programming languages suffers from similar conceptual and technical difficulties as encountered in the SQL domain, where these issues have become known as the *impedance mismatch*.

One standard solution with the goal of overcoming these issues is the Java Persistence API (JPA), which is a specification (latest version 2.1 from 2013)[3] for management of (primarily, but not limited to) relational data. Besides facilitating the mapping of Java entities to and from an underlying data store, it defines the criteria API which provides a programmatic, database-agnostic way for querying objects. Criteria queries are expressed over the classes and attributes of the domain model, and are thus independent of the specifics of the backend. JPA implementations perform the translation to corresponding queries supported by the respective backend.

[3] http://download.oracle.com/otndocs/jcp/persistence-2_1-fr-eval-spec/index.html

The essence of a *pure* JPA abstraction for RDF is the possibility to enable development against that data without having to deal with RDF and SPARQL specifics. The following advantages result from this: (i) Simplified consumption of RDF data from the Web of Data in Java applications by means of a declarative - rather than programmatic - mapping approach. Naturally, this limits the application to cases where the domain and RDF models are sufficiently similar for such mapping to exist. (ii) Unified querying over RDBs and triple stores via the criteria API, as well as higher flexibility in exchanging backends. (iii) De-silo-ification: Data silos of existing applications based on the JPA could be upgraded to use RDF stores and participate in the Web of Data, without *any* change in their application logic.

In this work, we make the following contributions towards enabling these benefits: (i) A JPA-based system design for enabling object/RDF (short: O/RDF) mappings, (ii) considerations for rewriting criteria queries to SPARQL via mappings, and (iii) a prototype implementation that enables querying over Java entities backed by RDF data. The prototype is available as Open Source[4] as the *mapper* module in our Jena-based Semantic Web toolkit. It is published on Maven Central[5] under the license is Apache 2 license.

The remainder is structured as follows: In Section 2, we present a simple example demonstrating a criteria query over an annotated Java class. In Section 3, we provide more details about the JPA and introduce important notions for rewriting them to SPARQL. Related work is summarized in Section 4. Afterwards, Section 5 describes the core design of our system, especially the aspect of establishing a mapping between Java object graphs and their corresponding RDF graph. In Section 6 we present our approach to rewriting criteria queries. Finally, we conclude in Section 7.

2 A Mapping and Criteria Query Example

The mapping of Java objects to and from RDF, as well as the criteria querying processing, is based on mapping information associated with classes. In this section we present an example based on DBpedia. Note, that in principle, mappings can be stored separately from classes, and multiple mappings can exist for a single class. Choosing the appropriate set of mappings is part of the O/RDF engine configuration. Our system supports a set of Java annotations for this purpose of which essential ones are demonstrated in Listing 1 and are described as follows: The *@DefaultIri* annotation is a non-obstrusive (i.e. requires no additional methods or attributes) way for specifying a rule how to generate IRIs for instances of the class. Its argument is a string in the *Spring Expression Language*. *@RdfType* causes RDF generated from entities of that class to include the corresponding *rdf:type* triple. This annotation also acts as a constraint during criteria query processing when requesting entities of that class. *@Iri* and *@IriNs* both associate an attribute with an RDF property, whereas *@IriNs* is a

[4] https://github.com/AKSW/jena-sparql-api/tree/master/jena-sparql-api-mapper

[5] http://search.maven.org/#search%7Cga%7C1%7Ca%3A%22jena-sparql-api-mapper%22

short-hand that constructs the property IRI by appending the attribute name to the given namespace. *@Lang* indicates that a String field maps to an RDF term with a language tag, as configured in the O/RDF engine. *@Datatype* needs to be specified if an attribute value's Java datatype differs from the one in the RDF model. A simple criteria query asking for all companies founded after 1950 having at least 10K locations is shown in Listing 2.

```
@RdfType("dbo:Company")
@DefaultIri("dbr:#{name}")
public class Company {
    @Lang @Iri("rdfs:label")                    private String name;
    @IriNs("dbo") @Datatype("xsd:gYear") private int foundingYear;
    @IriNs("dbo")                                private int numberOfLocations;
    /* ... */
}
```

Listing 1. An annotated Java *Company* domain class

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Company> cq = cb.createQuery(Company.class);

Root<Company> r = cq.from(Company.class);
cq.select(r)
    .where(cb.greaterThan(r.get("foundingYear"), 1950))
    .where(cb.greaterThanOrEqualTo(r.get("numberOfLocations"), 10000))
    .orderBy(cb.asc(r.get("foundingYear")))

List<Company> matches = em.createQuery(cq).getResultList();
```

Listing 2. A simple criteria query over the Company entity class

3 Preliminaries

3.1 Overview of the JPA

A criteria query comprises the following basic information. For brevity, we do not consider grouping, aggregates and sub-queries.

- The *result type*, which is the Java class being queried for. Most commonly, this is simply an entity class (such as Company), but it can also be the class of an entity's attribute (the company's name) or that of a computed value (the company's average number of locations).
- A set of *Query Roots* (short: root): A root always references an entity class and serves two purposes: (i) Roots introduce the initial sets of entities on which query evaluation operates. Evaluation of a criteria query conceptually constructs the cartesian product among the sets of entities referenced by the roots. (ii) Roots serve as starting points for navigation along *paths* of attributes. For example, a query root based on the *Company* class enables obtaining a path referencing the *foundingYear* attribute. It is important to note, that roots and paths *are* primitive expressions and can thus participate in compound ones.

- *Constraints*: A set of predicate expressions constraining the set of objects in the query result.
- *Orders*: A list of (expression, sort-direction) pairs.
- The *Selection*: An expression computing the final values of the result set (based on the query root's cartesian product). Often simply a root.
- *Distinct*: Removes duplicates from the result set.

The most important JPA components are:

- The *EntityManager* is the entry point for persistence-related operations on Java entities. It provides a standard interface for creating, reading, updating, and deleting entites(i.e. CRUD operations), and enables querying over them independently of the underlying data store. It provides the *getCriteriaBuilder* method which is the starting point for criteria query construction.
- The *CriteriaBuilder* is the factory for all criteria related constructs, namely criteria queries, compound selections, expressions, predicates, and orderings.

3.2 SPARQL Concepts and Roles

The process of rewriting criteria queries to SPARQL requires mapping entity classes and attributes to their counterparts in SPARQL. For this purpose, we introduce the following notions borrowed form description logics, and adapt them to SPARQL. Note, that a similar idea for translating OWL class expressions to SPARQL is presented in [1]. Let GP and V be the infinite sets of SPARQL graph patterns[6] and variables, respectively, and $vars$ be the function that yields a graph pattern's variables.

Definition 1. *A SPARQL Concept is a pair (gp, v) with $gp \in GP$ and $v \in vars(GP)$, and intentionally denotes a set of resources/individuals whose extension over an RDF graph is obtained by evaluating its graph pattern and projecting the stated variable.*

Definition 2. *A SPARQL role is defined as (gp, s, t) with $gp \in GP$ and $s, t \in vars(gp)$. Its evaluation over an RDF graph denotes a binary relation between a set of source and target resources.*

SPARQL roles are a powerful notion, as they enable relating resources to computed values, such as *({ ?s dbo:foundingYear ?x. BIND(year(?x) As ?o) }, ?s, ?o)*. This feature is necessary to e.g. correctly process the criteria query in Listing 2, where RDF terms of type *xsd:gYear* are mapped to Java integers. An *empty role* represents a zero-length path and is expressed as a role with an empty group graph pattern, and the same variable for source and target.

Definition 3. *SPARQL Role concatenation $r_1 \circ r_2 \rightarrow r_3$ yields a new role starting with the source variable of r_1 and the target one of r_2. The graph patterns of r_1 and r_2 are grouped into a new one, and a* FILTER *statement equating the target of r_1 with the source of r_2 is appended.*

[6] `https://www.w3.org/TR/sparql11-query/#GraphPattern`

4 Related Work

Well known implementations of the JPA specification are EclipseLink[7] (JPA's reference implementation), Hibernate[8] and Apache OpenJPA[9], which, to the best of our knowledge, do not feature RDF support. Yet, dedicated Java O/RDF mapping frameworks exist, which are based on either one of the two predominant Java RDF frameworks, namely Apache Jena[10] and Eclipse RDF4J[11].

Eclipse Komma[3][12] is an RDF4J-based framework, which provides its own non-JPA EntityManager API and distinguishes between interface and behaviour definitions. The latter implement one or more interfaces. Interfaces can carry RDF mapping information, similar to that in Listing 1. When loading a given RDF resource with Komma, the framework will, as of now, always yield a Java proxy implementing all suitable interfaces, whose method calls will delegate to all appropriate behaviors in customizable order. However, this approach makes it difficult to reuse third party code, as classes intended to act as behaviors may not be derived from a corresponding interface, and proxying, despite being a powerful feature, is known to sometimes cause subtle issues in regard to equality and inheritance checks. Historically, Komma evolved from the Alibaba[13] project, which in turn evolved from Elmo.

EmpireRDF[14] implements the JPA EntityManager interface and supports querying the RDFized data with SPARQL and SERQL. However, it does not feature support for criteria queries. Therefore, at present, none of the existing solutions facilitate a pure abstraction that enables querying the domain model without knowledge of the underlying RDF model.

5 System Architecture

In this section, we present the core design of our O/RDF system. Technically, the system is designed to account for two main functions: (i) Recursively serializing and de-serializing object graphs as RDF. For serialization, the process is initiated by requesting the state of a Java object to be written out as an RDF graph rooted in a given IRI. For de-serialization, the request is to load an IRI's RDF data as an instance of given class, whereas the returned object's actual type may be that of a subclass; e.g. a request for *Person* may yield an *Actor*. While this functionality does not differ substantially from related work, it is these parts of the system that hold RDF mapping information and need to be extended to support criteria query processing. (ii) Executing criteria queries on a SPARQL backend. This involves rewriting criteria queries to SPARQL and eventually deserializing IRIs as Java entities in order to construct the final result set.

[7] http://www.eclipse.org/eclipselink/
[8] http://hibernate.org/
[9] http://openjpa.apache.org/
[10] https://jena.apache.org/
[11] http://rdf4j.org/
[12] https://github.com/komma/komma
[13] https://bitbucket.org/openrdf/alibaba
[14] https://github.com/mhgrove/Empire

As part of the O/RDF mapping process, we need to determine the RDF graph corresponding to an entity. For this purpose, we introduce the notion of a *Resource Shape*, which is a specification to be evaluated over an RDF graph in regard to a given SPARQL concept. This yields for every resource matched by the concept the (possibly empty) RDF graph matching the shape, referred to as *shape graph*. Upon retrieval of an entity, its corresponding shape graph serves as the basis for populating its attributes. Upon storage of an entity, the set of added/removed triples is computed from comparing the shape graph at retrieval time to the RDF graph obtained from the entity's latest state. At present, resource shape specifications are built using our own API which provides methods for matching triples via ingoing and outgoing property paths. Relevant related work in this regard are the ongoing efforts on Shape Expressions (ShEx) [2] and the Shapes Constraint Language (SHACL)[15].

5.1 O/RDF Mapping Components

In this section, we describe the main components for de-/serializing object graphs from/to RDF graphs, depicted in Figure 1.

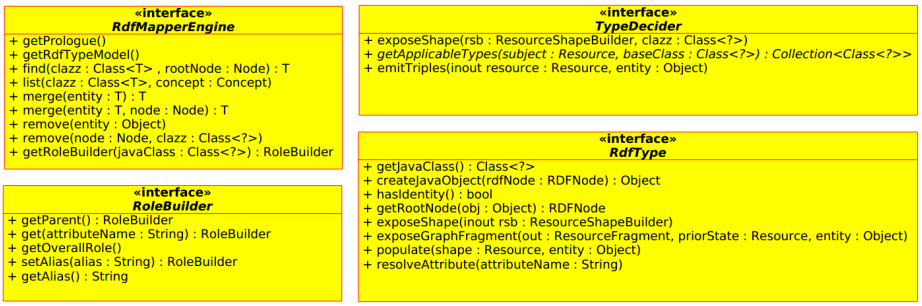

Fig. 1. Core components of the O/RDF mapping system

TypeDecider An IRI alone is insufficient for determining the appropriate set of corresponding candidate entity classes, as any class could act as a "view" over the resource's RDF data. The purpose of the *TypeDecider* is to narrow down this set of candidates – ideally to a single entity class. Note, that this functionality requires all entity classes to be known to the O/RDF system in advance. The TypeDecider supports exposing a resource shape for a given base class, whose results can be passed to the *getApplicableTypes* method in order to decide on the applicable sub-classes a resource can be loaded with. Also, for a given entity and its corresponding IRI, it can write out the triples needed to preserve the entity type in RDF.

RdfType is the core interface for establishing an O/RDF mapping for an individual Java class. The *getJavaClass* method returns that class and *createJavaObject* is used to create fresh, unpopulated instances of it. The latter method takes

[15] `https://www.w3.org/TR/shacl/`

an RDF term as argument in order to support instantiation of primitive/immutable Java types from RDF literals. The *getRootNode* method returns a Java object's RDF term which acts as the root in its RDF serialization. Some Java classes do not have an identity on their own, in which case *hasIdentity* returns false. For example, an instance of a *Collection* generally neither has an attribute nor an entry that uniquely identifies a specific instance. Yet, upon creating the RDF model, there needs to be an IRI that represents the collection in order to establish links to the contained items' corresponding RDF terms. If an object's class does not provide an IRI by itself, one is created based on the sequence of attribute names by which that object was reachable from an entity with an identity.

The *resolvePath* method is crucial for rewriting criteria queries to SPARQL. It returns for a given attribute name a *PathFragment* object, which holds the corresponding SPARQL role together with information for the *RdfMapperEngine* about how to resolve sub-paths.

The *exposeShape* method yields the *RdfType*'s resource shape. The corresponding shape graph can be passed to the *populate* method, in order to partially update a given entity's state and obtain a set of references which need further resolution. A reference comprises the information of which IRI needs to be resolved as an entity of which Java class, together with a callback function that updates the given entity's state with the result of the resolution.

The *exposeGraphFragment* method converts an entity's state to an RDF graph fragment and is thus the opposite operation of *populate*. The exposed fragment comprises an RDF graph together with a mapping of which of its resources correspond to entities for which further RDF graph fragments can be obtained.

RDFMapperEngine provides the essential functionality for retrieval and storage of entities and handles all recursive aspects of RDF de-/serialization and role construction based on the registered *RdfTypes*. Notably, it provides the *RoleBuilder* facade for resolving *paths* of attribute names to SPARQL roles. The *RDFMapperEngine* is the core of our JPA *EntityManager* implementation.

6 Rewriting Criteria Queries

Here, we outline our approach for rewriting criteria queries to SPARQL. The fundamental operation is to rewrite primitive criteria expressions, i.e. roots, paths and constants, to SPARQL. Let P be an initially empty list of graph patterns, and *rewrite* a function that yields for a given criteria expression a SPARQL expression. Then $rewrite(path, P)$ obtains the SPARQL role from *RDFMapperEngine*, adds the role's graph pattern to P and returns the target variable as a SPARQL expression. Alias names become SPARQL variable names, and during rewriting, every criteria expression is assigned a fresh unique alias if none was provided at query construction time. As arithmetic, (in)equality, and conditional criteria expressions have direct counterparts in SPARQL, their rewrite is: $rewrite(op_{criteria}(a_1, \ldots, a_n), P) \rightarrow op_{sparql}(rewrite(a_1, P), \ldots, rewrite(a_n, P))$

Let Q be the target SPARQL query. The essential rewriting steps are:

- Add every query root's corresponding SPARQL concept graph pattern to Q.
- Rewrite the constraint expressions and add them as FILTERs to Q.
- Rewrite the sort condition expressions and add their graph patterns as OPTIONAL patterns to Q.
- Add the graph patterns of selection expressions as OPTIONAL patterns.
- Apply DISTINCT, LIMIT and OFFSET of the criteria query directly to Q.

Finally, the criteria query result set is constructed by executing the SPARQL query and using each solution binding to retrieve entities from the *RdfMapperEngine* according to the criteria query's selection. Listing 3 shows the rewrite of Listing 2.

```
SELECT DISTINCT ?s {
  ?s a dbo:Company .
  ?s dbo:foundingYear ?a      . BIND(year(?a) As ?x) . FILTER(?x > 1950)
  ?s dbo:numberOfLocations ?y . FILTER(?y >= 10000)
  OPTIONAL { ?s dbo:foundingYear ?b . BIND(year(?b) As ?z) }
} ORDER BY ASC(?z)
```

Listing 3. The example criteria query translated to SPARQL

7 Conclusion and Future Work

In this submission, we (i) presented a system architecture for object/RDF mappings, (ii) outlined notions for rewriting JPA criteria queries to SPARQL, and (iii) provide an Open Source prototype implementation. We demonstrate by example that our approach enables querying over Java domain models without the need to be aware of RDF and SPARQL. For certain use cases, this can greatly simplify querying, consumption, creation and modification of RDF data in Java applications. Our approach to translating criteria queries to SPARQL can be naturally complemented with arbitrary SPARQL (federation) engines in order to facilitate querying over the Web of Data. In the future we will extend the feature set, work on a more rigorous formalization and clarify semantic aspects. Directions for future research in regard to O/RDF systems are query optimization and performance analyses.

Bibliography

[1] S. Bin, L. Bühmann, J. Lehmann, and A.-C. Ngonga Ngomo. Towards SPARQL-based induction for large-scale RDF data sets. In *ECAI 2016 - Proceedings of the 22nd European Conference on Artificial Intelligence*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 1551–1552. IOS Press, 2016.

[2] S. Staworko, I. Boneva, J. E. Labra Gayo, S. Hym, E. G. Prud'hommeaux, and H. Solbrig. Complexity and expressiveness of shex for rdf. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 31. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[3] K. Wenzel. Komma: An application framework for ontology-based software systems. *Semantic Web–Interoperability, Usability, Applicability*, 2010.