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. It comprises a set of Java interfaces, annotations, and the JPA query language (JPQL) and thus provides a framework for uniform persistence and retrieval of Java objects using various backends. These interfaces include the *Criteria API* for building queries programmatically, allowing queries to be written once against a Java domain model, and executing them on any supported backend, which bears the potential of higher productivity in enterprise-grade software development due to less repetitive – and prone to errors – code in the data tier. Our contributions comprise (i) a lightweight 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 graph patterns, and (iii) a prototype implementation.

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

1 Introduction

The Java Persistence API (JPA) is a specification (latest version 2.1 from 2013)[3] for management of (primarily, but not limited to) relational data. Besides facilitating object-relational mapping, it defines the JPA query language (JPQL) and the criteria API which provide a database-agnostic way for persisting and querying Java objects. However, so far there is a lack of support for RDF backends. As a motivating example, consider Listing 1 which shows one of the SQL queries used in the backend of the *spring-batch*[4] workflow engine. If one wanted to track job status information directly in RDF, it is certainly possible to craft equivalent SPARQL queries against an appropriate RDF model. However, porting queries to different backends can be a tedious and error prone task. Further, static SPARQL queries would be tied to one specific set of vocabulary terms, making minor adoptions difficult. In contrast, with modern O/RM tooling, writing queries can be achieved with code similar to Listing 2.

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

[4] `http://projects.spring.io/spring-batch/`

```
SELECT JOB_EXECUTION_ID, START_TIME, END_TIME, STATUS, EXIT_CODE,
    EXIT_MESSAGE, CREATE_TIME, LAST_UPDATED, VERSION,
    JOB_CONFIGURATION_LOCATION from JOB_EXECUTION
where JOB_INSTANCE_ID = ? order by JOB_EXECUTION_ID desc
```

Listing 1. An SQL-query used in spring-batch's JobExecution DAO

```
CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<JobExecution> cq = cb.createQuery(JobExecution.class);
Root<JobExecution> r = cq.from(JobExecution.class);
cq.select(r).where(
    cb.equal(r.get("jobInstanceId"), jobInstanceId))
  .orderBy(cb.desc(r.get("executionId")));

TypedQuery<JobExecution> query = entityManager.createQuery(cq);
List<JobExecution> tq = query.getResultList();
```

Listing 2. Corresponding criteria query

Some advantages of an object-RDF mapper, from here on referred to as O/RDF, are: No code required to manually map RDF triples to objects. As code is written against the Java model, and is thus independent of the backend which stores the objects, legacy systems already using the JPA can be transparently turned into Semantic Web applications.

Our contributions are as follows: (i) A lightweight, yet flexible, system design for de-/serializing object graphs as RDF, and (ii) formal aspects of rewriting criteria queries to SPARQL via the mappings, and (iii) a prototype implementation that enables querying Java entities backed by RDF data via the JPA criteria API. The prototype is available as Open Source[5] as the *mapper* module in our Jena-based Semantic Web toolkit. The license is Apache 2, and maven artifacts are published on maven central[6].

The remainder is structured as follows: In Section 2, we present preliminaries in regard to the used terminology and the JPA. Related work is summarized in Section 3. Afterwards, Section 4 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 5 we present our approach to rewriting criteria queries. Finally, we conclude in Section 6.

2 Preliminaries

In this section, we give a brief overview of relevant terms, important components of the JPA, and essentials of the criteria API.

Terminology : An entity is a *lightweight persistence domain object*. An *entity class* is an *ordinary Java class that is marked as having the ability to represent objects in the database*. Entities can be identified as well as referenced by an (IRI, class) pair. A reference matches all entities with the same IRI that are subclasses of the given class.

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

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

JPA Criteria API The most important JPA components are:

- The *EntityManager* is the entry point for persistence-related operations on Java entities. It provides standard interfaces for creating, reading, updating, deleting (i.e. CRUD operations), and querying over entities independent of the underlying data store. It provides the *getCriteriaBuilder* method which is the starting point for criteria query construction.
- The *CriteriaBuilder* is used to construct criteria queries, compound selections, expressions, predicates, orderings[7].
- *Expressions*: Primitive expressions are formed from *paths* over attribute names of the corresponding entity classes. Compound expressions are built using the CriteriaBuilder and include unary, arithmetic, comparison, conditional operators.

A criteria query comprises the following basic information:

- The *result type*.
- A set of *Query Roots*: A root represents the start of a navigation along paths of attributes from a referenced entity.
- *Constraints*: A set of predicate expressions constraining the set of entities.
- *Orders*: A list of (expression, sort-direction) pairs.
- *Distinct*: Removes duplicates from the result set.

For brevity, we do not consider grouping/aggregates, sub-queries and selections. The special aspect of the criteria API is that primitive expressions are formed by paths of attribute names created from roots.

3 Related Work

Well known implementations of the JPA specification include EclipseLink[8] (JPA's reference implementation), Hibernate[9] and Apache OpenJPA[10], which, to the best of our knowledge, do not feature RDF support. Yet, dedicated Java/RDF mapping frameworks exist, which are usually based on one of the two predominent Java RDF frameworks, namely Apache Jena[11] and Eclipse RDF4J[12] (formerly Sesame).

Eclipse Komma[4][4][13] is an RDF4J-based framework, which provides its own 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 3. When loading a given RDF

[7] http://docs.oracle.com/javaee/6/api/javax/persistence/criteria/CriteriaBuilder.html
[8] http://www.eclipse.org/eclipselink/
[9] http://hibernate.org/
[10] http://openjpa.apache.org/
[11] https://jena.apache.org/
[12] http://rdf4j.org/
[13] https://github.com/komma/komma

resource with Komma, it will yield a Java proxy implementing all suitable interfaces, whose method calls will delegate to all appropriate behaviors. Historically, Komma evolved from the Alibaba[14] project, which in turn evolved from Elmo.

EmpireRDF[15] implements the JPA EntityManager interface and supports querying the RDFized data with SPARQL and SERQL. However, it does not feature support for JPQL or criteria queries.

4 System Architecture

In this section, we present the core design of our O/RDF system. Similar to Komma and EmpireRDF, we introduce annotations for controlling RDF specifica of the mapping not covered by the standard JPA annotations. 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 a subclass of a given class. (ii) Executing criteria queries on a SPARQL backend. This involves rewriting constraints expressed with the criteria API to SPARQL, in order to identify the set of IRIs of the entities which form the query's result set and thus need to be de-serialized.

In order to retrieve the appropriate RDF graph fragment to populate the attributes of an entity, we introduce the notion of a *Resource Shape*, which expresses a pattern for a set of (nested) RDF triples reachable from a resource. Relevant related work in this regard are the ongoing efforts on Shape Expressions (ShEx) [3] and the Shapes Constraint Language (SHACL)[16].

4.1 Annotations

Annotations are a simple approach to tie a mapping directly to a class, such as in Listing 3. Note, that in principle, O/RM mappings are independent entities, and thus multiple mappings may exist for a set of classes. Choosing the appropriate set of mappings is part of the O/RM engine configuration. The annotation *@DefaultIri* accepts an expression string in the *Spring Expression Language* (SpEL) to dynamically generate a default IRI for instances of an entity class. *@RdfType* will cause the generated RDF to to include a triple that associates the entity's IRI with the stated.

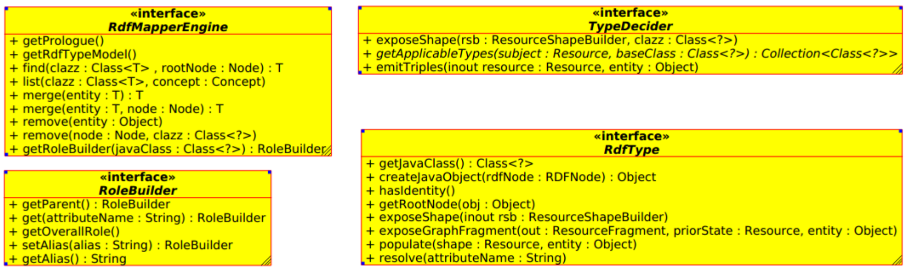
```
@RdfType(":JobExecution")
@DefaultIri(":job-execution#{instanceId}-#{executionId}")
public class JobExecution {
    @Iri(":instanceId") private String instanceId;
    @Iri(":executionId") private long executionId;
    @Iri(":config") private Map<String, Object> params;
}
```

Listing 3. RDF Mapping annotations of a class

[14] https://bitbucket.org/openrdf/alibaba
[15] https://github.com/mhgrove/Empire
[16] https://www.w3.org/TR/shacl/

4.2 Components for O/RDF mapping

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



```
«interface»
RdfMapperEngine
+ getPrologue()
+ getRdfTypeModel()
+ find(clazz : Class<T> , rootNode : Node) : T
+ list(clazz : Class<T>, concept : Concept)
+ merge(entity : T) : T
+ merge(entity : T, node : Node) : T
+ remove(entity : Object)
+ remove(node : Node, clazz : Class<?>)
+ getRoleBuilder(javaClass : Class<?>) : RoleBuilder

«interface»
RoleBuilder
+ getParent() : RoleBuilder
+ get(attributeName : String) : RoleBuilder
+ getOverallRole()
+ setAlias(alias : String) : RoleBuilder
+ getAlias() : String

«interface»
TypeDecider
+ exposeShape(rsb : ResourceShapeBuilder, clazz : Class<?>)
+ getApplicableTypes(subject : Resource, baseClass : Class<?>) : Collection<Class<?>>
+ emitTriples(inout resource : Resource, entity : Object)

«interface»
RdfType
+ getJavaClass() : Class<?>
+ createJavaObject(rdfNode : RDFNode) : Object
+ hasIdentity()
+ getRootNode(obj : Object)
+ exposeShape(inout rsb : ResourceShapeBuilder)
+ exposeGraphFragment(out : ResourceFragment, priorState : Resource, entity : Object)
+ populate(shape : Resource, entity : Object)
+ resolve(attributeName : String)
```

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 The *RdfType* interface provides the essential high-level methods for mapping classes to RDF. RdfType is the core interface for establishing a mapping of a single Java class, retrievable by *getJavaClass*, to the RDF model. Newly unpopulated instances can be created from an RDFNode using the `createJavaObject` class. Java primitive and composite types roughly correspond to RDF literals and resources, respectively. An RDFNode and can be a literal or a resource. In the former case, the result is expected to be the corresponding (boxed) Java primitive type, such as 5l returned for the argument 58sd:long. For resources, the RDF type is expected to yield an unpopulated entity.

Some Java classes do not have an identity on their own, in which case `hasIdentity` returns false. For example, an instance of a `Collection` usually neither has an attribute nor an entry that uniquely identifies this specific instance. Yet, there needs to be an IRI that represents the collection in order to retain links to the IRIs of the contained items. If an object's class does not provide an IRI by itself, we create one based on the owning entity's IRI and the attribute names by which that object was reachable from an entity. For example, the IRI of the `Map` in Listing 3 will be the default IRI of that class with the attribute name *params* appended.

The *resolvePath* method returns for a given property available in the class the corresponding SPARQL role (see Section 5), encapsulated in the PathFragement object.

In regard to querying, the resolvePath method is crucial for compiling the WHERE clause of a criteria query to SPARQL, the exposeShape denotes which RDF graph to retrieve, and the populate/exposeFragment methods are required to set a Java object's state according to the state of a (retrieved) RDF graph, serialize the current state as triples possibly by reusing RDF terms from its initial RDF graph.

RdfType is *only* responsible for writing and reading triples relevant to the class it represents. Recursive resolution is handled by the engine based on the information returned by *RdfType*, concretely

– when reading RDF, which RDF terms in the graph fragment, that matched the shape, need to be further resolved to Java objects of which base class.
– when writing RDF, which RDF terms in the output need to be substituted with RDF nodes obtained for Java objects.

RDFMapperEngine : The low-level interface for managing entities for which a JPA wrapper is provided.

RoleBuilder : The role builder is capable of mapping paths of attribute names to SPARQL roles.

5 Rewriting Criteria Queries

Here, we introduce SPARQL based notions for formally capturing the rewriting steps. We first borrow the notions of concepts and roles form description logics and adapt them to SPARQL. A similar idea is presented in [1], however we make more considerations especially in regard to handling variables. Subsequently, we introduce operations for navigation between SPARQL concepts via roles, role chains, and sorting of concepts. Finally, we present the rewriting steps.

Let there be the set of variables V and the set of graph patterns[17] GP. For brevity, we assume the reader is familiar with evaluation of graph patterns according to the SPARQL specification.

Definition 1. *Projected Graph Patterns (PGPs): A projected graph pattern is a pair $pgp := (gp, W)$, with gp a graph pattern and $W \subset V$ a set of variables appearing in gp, which we refer to as* distinguished variables. *All other variables are* undistinguished. *This additional information about a gp's variables is subsequently used to determine which variables to rename / leave fixed when performing operations on them.*

Definition 2. *SPARQL Concepts: A SPARQL concept is a pair (pgp, v) with $v \in V$. SPARQL concepts thus denote sets of resources/individuals expressed as a PGP with one of its variables tagged.*

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

Definition 3. *Ordered SPARQL Concepts: An ordered concept expresses an ordering over a set of resources identified by a concept. We define it as* (c, SCS) *with* c *a concept and* SCS *a sequence of SPARQL sort conditions in terms of variables in c. We further introduce an operator applyOrder* : $(\mathcal{O}, \mathcal{C}) \rightarrow \mathcal{O}$, *which orders the items in* C *according to their order in* $O.order$.

Definition 4. *SPARQL role: A SPARQL role is defined as* (pgp, s, t) *with* $s, t \in V$. *Its evaluation therefore denotes a binary relation between a set of source and target resources. Note, that this is a powerful notion, as it e.g. enables relating resources to computed values, such as* `( ?s rdfs:label ?o. Bind(lang(?o) As ?x) , ?s, ?x)`. *An* empty role *represents a zero-length path and is expressed as a role having a PGP with an empty group graph pattern, and the same variable for source and target.*

Concept conjunction : $C \cap C \rightarrow C$: yields a new concept by equating their result variable, combining their graph patterns, and renaming any common undistinguished variables.

Role concatenation : $R_1 \circ R_2 \rightarrow R$ yields a new role starting with the source variable of r_1 and the target on of r_2, thereby renaming common undistinguished variables.

Navigation : $C.R \rightarrow C$: $c.r$ Yields a new concept by traversing the role r from c. Again, common undistinguished variables are renamed.

Ordering a concept : $applyOrder(O, C) \rightarrow O$: An ordered concept can be used to sort another concept by appropriately combining their graph patterns and sort conditions. Ordering does not change the (multi-)set of resources matched by the original concept.

Optimizing graph patterns : Graph patterns resulting from the operations may contain redundant triple patterns and expressions which however differ by the variables. For example, a concept *({?s rdfs:label ?x, ?y}, ?x)* could be optimized, if $?y$ was substituted for $?x$. The technique that accomplishes this was first described in [2].

Above definitions provide the tooling required to assemble SPARQL queries from criteria expressions. The fundamental operation is converting paths of attribute names to SPARQL roles, i.e. graph patterns with a designated target variable. For example, resolving an attribute path *address.street* on a *Person* class to a role is accomplished by requesting the RoleBuilder *rb* for Person from the RdfMapperEngine, and invoking *rb.get("address").get("street")*. The engine forwards these requests to the appropriate RdfTypes and assembles the overall role. Compound criteria expression can then be directly converted to SPARQL expressions over these role's target variables.

The remaining steps are to compile a criteria query to a corresponding ordered concept matching the IRIs of the desired entities and loading the entities as necessary. This ordered concept is comprised of:

- The SPARQL concept from the type decider for the requested entity class (and those used as roots).
- The compilation of expressions from the WHERE part.
- The compilation of the criteria sort conditions into an ordered concept.
- Appropriate use of DISTINCT, LIMIT and OFFSET in the generated SPARQL graph pattern, in accordance with the given criteria query.

Finally, for each resource matching the compiled SPARQL concept, the corresponding entity is retrieved from the EntityManager using the query's result class.

6 Conclusion and Future Work

In this submission, we (i) outlined a system architecture for object/RDF mappings, (ii) introduced formal notions for SPARQL-based concepts, orderered concepts, roles, and relevant operators in order to rewrite JPA criteria query constraints to SPARQL, and (iii) provide an Open Source prototype implementation. This effectively enables uniform querying of Java domain models backed by (SPARQL accessible) RDF data. In the future, we will add support for more criteria query features, clarify certain semantics, such as deleting entities whose corresponding triples back other entities, and performance will be evaluated and optimized, such as by batching lookups of multiple resources by their resource shape.

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] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90. ACM, 1977.

[3] 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.

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