Boosting RDF Adoption in Ruby with Goo

Manuel Salvadores, Paul R. Alexander, Ray W. Fergerson,
Natalya F. Noy, and Mark A. Musen

Stanford Center for Biomedical Informatics Research
Stanford University, US
{manuelso,palexander,ray.fergerson,noy,musen}@stanford.edu

Abstract. For the last year, the BioPortal team has been working on a new iteration that will incorporate major modifications to the existing services and architecture. As part of this work, we transitioned BioPortal to an architecture where RDF is the main data model and where triple stores are the main database systems. We have a component (called "Goo") that interacts with RDF data using SPARQL, and provides a clean API to perform CRUD operations on RDF stores. Using RDF and SPARQL for a real-world large-scale application creates challenges in terms of both scalability and technology adoption. In BioPortal, Goo helped us overcome that barrier using the technology that developers were familiar with, an ORM-alike API.

Keywords: SPARQL, RDF, ORM, Ontologies

1 Why Goo? Why a Framework?

BioPortal, developed in our laboratory, provides access to semantic artifacts such as ontologies [9]. Our team has developed a new iteration of the BioPortal REST API and related infrastructure. The most significant architectural change is the replacement of the backend systems with an RDF triplestore. This single RDF triplestore replaces a variety of custom database schemas that were used to represent ontologies originated in different languages.

The BioPortal REST API provides search across all ontologies in its collection, a repository of automatically and manually generated mappings between classes in different ontologies, ontology reviews, new term requests, and discussions generated by the ontology users in the community [9]. Most importantly, our API provides uniform access to the terminologies regardless of the language used to develop them. Naturally, we did not expect the majority of developers on our team and others who access the REST API to understand which specific SPARQL query to use to access this complex information. Rather, it became much more efficient to abstract the SPARQL queries into an API that operates at the resource level. Goo (which stands for "Graph Oriented Objects") is the library that we developed for this purpose. In many ways, Goo contains characteristics of traditional Object-Relational Mapping libraries (ORMs). ORMs are widely used to handle persistency in relational databases and to provide an abstraction over the physical structure of the data and the raw, underlying SQL queries. They also help to map data between relational models and object-oriented programming languages. Popular ORMs include Hibernate, ActiveRecord and SQLAlchemy for Java, Ruby and Python respectively. Goo is therefore an ORM specifically designed to work with SPARQL backends. The Goo library frees developers from thinking about the intricacies of SPARQL, while still exposing the power of RDF's ability to interconnect data.

The driving requirements for our design are the following:

Abstraction: Though BioPortal uses Semantic Web technologies, not all of the BioPortal development team has been exposed to RDF and SPARQL. This situation is probably common in many other development teams. At the same time, most professional developers have dealt extensively with ORMs–like Hibernate and ActiveRecord–and most developers feel very comfortable working with them.

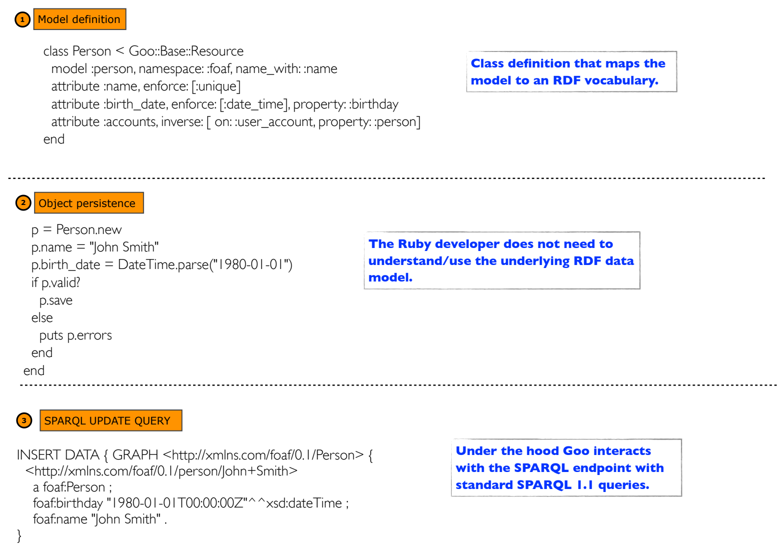
Scalability: Our data-access layer must be aware of the query patterns for which the triplestore performance excels and must try to rely on those patterns as much as possible.

Flexibility: The schemaless nature of triplestores supports heterogeneity very well. Our store contains 2,541 different predicates, but the application needs to provide special handling for only a small portion of those [7]. Our framework needs to be flexible enough to let the developer choose what data attributes get included in the retrieval.

A number of libraries for different platforms offer ORM-like capabilities for RDF and SPARQL. Jenabean uses Jena's flexible RDF/OWL API to persist Java Beans [8]. But Jenabeans approach is driven by the Java object model rather than an OWL or RDF schema. A number of tools use OWL schemas to generate Java classes [1, 2]. These tools enable Model Driven Architecture development, but do not provide support for triple stores. For Python, RDFAlchemy provides an object-type API to access RDF data from triplestores. It supports both SPARQL backends and Python RDFLib memory models [5]. ActiveRDF is a library for accessing RDF data from Ruby programs. It can be used as data layer in Ruby-on-Rails, similar to ActiveRecord, and it provides an API to build SPARQL queries programmatically [6]. The SPIRA project, also for Ruby, provides a useful API for using information in RDF repositories that can be exposed via the RDF.rb Ruby library [4, 3]. Because we use Ruby in the new BioPortal platform, we considered SPIRA as our ORM. However, SPIRA's query strategy was not built to handle very large collections of artifacts and the query API did not allow for the complex query construction that BioPortal needs.

2 Goo's API in a Nutshell

This section briefly introduces the Goo API. In this section, and the rest of the paper, we describe the API using a subset of BioPortal models that are complex enough to help us outline Goo's capabilities.


1 Model definition
Class definition that maps the model to an RDF vocabulary.
2 Object persistence
The Ruby developer does not need to understand/use the underlying RDF data model.
3 SPARQL UPDATE QUERY
Under the hood Goo interacts with the SPARQL endpoint with standard SPARQL 1.1 queries.

```
class Person < Goo::Base::Resource
  model :person, namespace: :foaf, name_with: :name
  attribute :name, enforce: [:unique]
  attribute :birth_date, enforce: [:date_time], property: :birthday
  attribute :accounts, inverse: [ on: :user_account, property: :person]
end
```

```
p = Person.new
p.name = "John Smith"
p.birth_date = DateTime.parse("1980-01-01")
if p.valid?
  p.save
else
  puts p.errors
end
end
```

```
INSERT DATA { GRAPH <http://xmlns.com/foaf/0.1/Person> {
  <http://xmlns.com/foaf/0.1/person/John+Smith>
   a foaf:Person ;
   foaf:birthday "1980-01-01T00:00:00Z"^^xsd:dateTime ;
   foaf:name "John Smith" .
}
```

Fig. 1. The top of this figure is an example of a Goo model definition. The settings in this model establish object validations and how this model is represented in RDF. For more details on each of these settings see the project documentation page at http://ncbo.github.io/goo/. The second part of this figure is a script that shows how we achieve persistence like similar ORM-alike libraries.

The following set of models is mentioned in the remainder of the paper: User (describes a user profile), Role (describes different roles of users in the application, such as an administrator), Note (describes comments on ontologies provided by users), Ontology (describes the object that represents an ontology entry in our repository) and OWLClass. We do not provide a fully detailed schema for these objects; each example is self-contained and the relations between objects should be clear to the reader. The full documentation for the Goo API is available at `http://ncbo.github.io/goo/`.

Goo models are regular Ruby classes. To enable RDF support each model needs to extend the `Goo::Base::Resource` class. Figure 1 gives a brief description of how models get defined. In the same figure it is shown how Ruby developers can save and validate the object without having to deal with RDF and/or SPARQL.

Once objects are defined, Goo provides a framework for creating, saving, updating and deleting object instances. Goo assures uniqueness of RDF IDs in collections and tracks modified attributes in persisted objects. The DSL allows us to provide both validation rules and define how objects are interlinked.

Ruby is a typeless language and thus developers can assign arbitrary value types to variables and attributes (i.e: the language does not enforce the assignment of `Date` values to a property that should only accept `Date` objects). We rely on Goo to perform these operations automatically and transparently, notifying when a validation fails. Goo incorporates multiple built-in validations for data types like email, URI, integer, float, date, etc. Moreover, the framework can be extended by using Ruby lambdas which can be needed to perform custom validations. For example, a custom ISBN format validation can be included in the set of validations for a model with the following `attribute` definition:

```
attribute :isbn, enforce: [ lambda { |self| isbn_valid?(self.isbn) } ]
```

2.1 Querying: From Graphs of Triples to Graphs of Objects

Goo's most important feature is its flexible query API. The API allows retrieving individual objects, their attributes, collections, and so on.

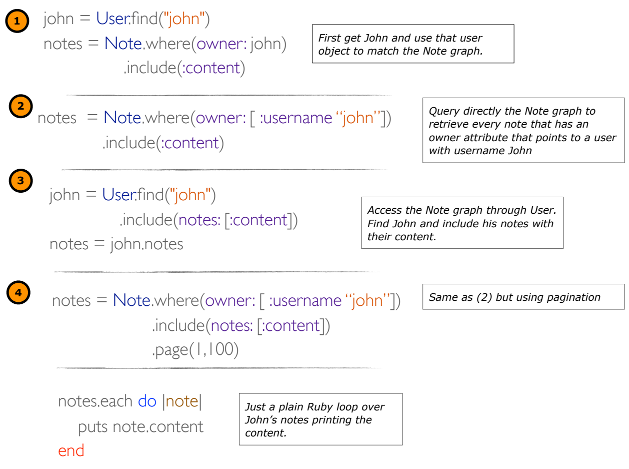
Retrieving individual objects One can retrieve single resource instances using `Resource.find`. This call is useful when querying by unique attributes or when the URI that identifies a resource is known.

Retrieving object attributes By default, none of the query API calls attach any attribute values to the instance object that they return. If we try to access an attribute that has not been included (i.e., retrieved from the triplestore), Goo throws an `AttributeNotLoaded` exception. Our design always defaults to strategies that imply minimum data movements. This strategy improves efficiency by retrieving only the attributes that the application cares about. Data attributes are loaded into objects by using the `include` command (Figure 2).

Incremental Object Retrieval We have encountered situations in which one might not know exactly what attributes need to be loaded in an object. Goo allows incremental, in-place retrieval of attributes. An array of already loaded objects can be populated with more attributes. This operation is in-place because Goo will not create a new array of objects but will populate the objects that are passed into the query via the `models` call.

```
Users.where.models(users).include(notes: [:content])
```

Pagination Our REST API outputs large collections of data and in some cases we have to implement pagination over the responses. Pagination in SPARQL, with `LIMIT` and `OFFSET`, works at the triple level but it is not trivial for non SPARQL experts to develop the queries that retrieve a paginated collection of items. Goo provides capabilities that abstract the intricacies of triple level pagination and leverages this capability to the Ruby objects. Every query in Goo can be paginated, the underlying SPARQL query uses SPARQL `LIMIT` and `OFFSET` to assure low data transfers and minimun object instantiation. A Goo API paginated call is shown in Figure 2.4.



1
```
john = User.find("john")
notes = Note.where(owner: john)
            .include(:content)
```
First get John and use that user object to match the Note graph.

2
```
notes  = Note.where(owner: [ :username "john"])
           .include(:content)
```
Query directly the Note graph to retrieve every note that has an owner attribute that points to a user with username John

3
```
john = User.find("john")
            .include(notes: [:content])
notes = john.notes
```
Access the Note graph through User. Find John and include his notes with their content.

4
```
notes = Note.where(owner: [ :username "john"])
            .include(notes: [:content])
            .page(1,100)
```
Same as (2) but using pagination

```
notes.each do |note|
   puts note.content
end
```
Just a plain Ruby loop over John's notes printing the content.

Fig. 2. Four different ways to retrieve John's notes. 1: First retrieve the user and then the notes. 2: Match the graph with a slightly more complex pattern. 3: Extract the notes by including them in a User instance. 4: same as (2) but uses pagination.

Creating complex queries The API also allows for more complex query definitions. One can combine calls with `or` and `join`, and with these we internally construct SPARQL `UNION` blocks that can be combined with SPARQL joins. Range queries can be also defined using the `Filter` object and the `filter` method. All of these operations can be combined to build complex queries.

```
filter_on_created =
      (Goo::Filter.new(:created) > DateTime.parse('2011-01-01'))
      .and(Goo::Filter.new(:created) < DateTime.parse('2011-12-31'))

Users.where(notes: [ ontology: [ acronym: "SNOMEDCT"] ])
            .or(notes: [ ontology: [ acronym: "NCIT"] ])
            .join(notes: [ :visibility [ code: "public"]])
            .filter(filter_on_created)
            .include(:username, :affiliation)
```

The Ruby code above represents a Goo query that retrieves the list of users, with their `:username` and `:affiliation`, that submitted notes to the ontologies `"SNOMEDCT"` or `"NCIT"` and these notes have visibility code `"public"`. In addition, a filter is created to filter users to just the ones that were created in the system between a range of dates. This query has an extra complexity, the `notes` attribute in `User` is defined as an inverse attribute. Goo is able to reverse the SPARQL query patterns to match the graph with the correct pattern directionality. The filtering implementation also allows for retrieval of nonexistent graph patterns. To retrieve the list of users that never submitted a note we simply use the `unbound` call in `Filter`.[1]

[1] See usage of `Filter.unbound` at `http://ncbo.github.io/goo/`

3 Goo's Query Strategy

Different query strategies can lead to starkly different performance in SPARQL. A triplestore might have an efficient query implementation, but if our application, for example, moves data around a lot, our queries will not perform well. Indeed, one often hears complaints about the performance of SPARQL engines whereas the real issue is the client who is not using SPARQL efficiently. Our key rationale with implementing query strategies in Goo is to provide a layer that ensures efficient access and query decomposition for SPARQL without the developer having to worry about it.

Goo's strategy navigates the graph of included patterns recursively. The first query focusses on constraining the graph and retrieving attributes that are adjacent to the resource type. To retrieve data attributes located more than one hop away, Goo runs additional queries—as many of these queries are types of resources that are involved in the retrieval request. As a result, when a developer uses Goo to request attributes of dependent resources, Goo will decompose the request into multiple queries.

Goo traverses the graph patterns recursively using a Depth First Search (DFS); it focuses on individual resource types in each step. Figure 3 (right side) shows how we chain these queries together with SPARQL `FILTER`s that join sequences of `OR` operations. These filters help each subsequent query to retrieve only attributes for the dependent models and not the entire collection.[2]

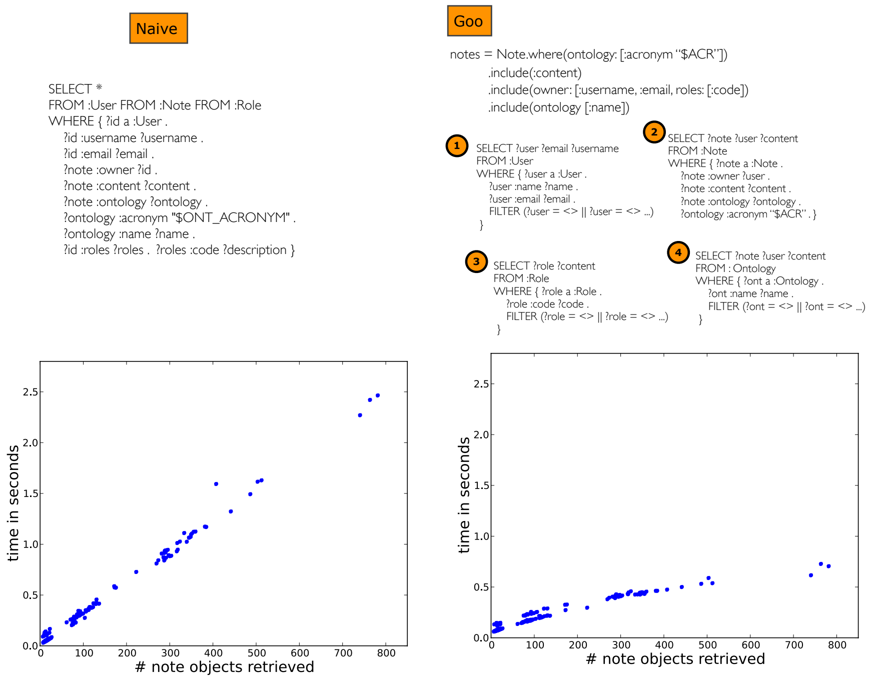
Consider the following example. In BioPortal, a user can attach notes to ontologies. A note description links to a user (author of the note) and the ontology that the note refers to. Our testing dataset contains 400 ontologies, 10K notes, 900 users and 10 roles. We have intentionally skewed our testing dataset so that the top 10% ontologies account for 55% of the notes—the distribution that reflects the actual state of affairs in BioPortal. Figure 3 highlights the performance gain when retrieving notes for each ontology in BioPortal using Goo. In the Naive approach performance degrades when we start projecting, into tabular form, n-ary relations that are hidden in the graph. We have seen that it is often the case that hand-written SPARQL queries retrieve too much data at once, and this can cause combinatorial data explosions due to the hidden n-ary relations.

As it can be seen in Figure 3, we can demonstrate the difference in the performance of different query strategies even on this small dataset. Goo's strategy will recursively navigate different types of objects in the retrieval, avoiding combinatorial explosions.

4 Discussion

Traditional software development and database development has a long history of reliable frameworks to access backend systems. The majority of Semantic Web

[2] For this experiment we used 4store 1.1.5 in a cluster setup with 8 backend nodes.



```
SELECT *
FROM :User FROM :Note FROM :Role
WHERE { ?id a :User .
    ?id :username ?username .
    ?id :email ?email .
    ?note :owner ?id .
    ?note :content ?content .
    ?note :ontology ?ontology .
    ?ontology :acronym "$ONT_ACRONYM" .
    ?ontology :name ?name .
    ?id :roles ?roles .  ?roles :code ?description }
```

```
notes = Note.where(ontology: [:acronym "$ACR"])
        .include(:content)
        .include(owner: [:username, :email, roles: [:code])
        .include(ontology [:name])
```

```
SELECT ?user ?email ?username
FROM :User
WHERE { ?user a :User .
    ?user :name ?name .
    ?user :email ?email .
    FILTER (?user = <> || ?user = <> ...)
 }
```

```
SELECT ?note ?user ?content
FROM :Note
WHERE { ?note a :Note .
    ?note :owner ?user .
    ?note :content ?content .
    ?note :ontology ?ontology .
    ?ontology :acronym "$ACR" . }
```

```
SELECT ?role ?content
FROM :Role
WHERE { ?role a :Role .
    ?role :code ?code .
    FILTER (?role = <> || ?role = <> ...)
 }
```

```
SELECT ?note ?user ?content
FROM : Ontology
WHERE { ?ont a :Ontology .
    ?ont :name ?name .
    FILTER (?ont = <> || ?ont = <> ...)
 }
```

Fig. 3. Left-hand side shows a SPARQL query template that returns all notes and user information for a ontology in BioPortal. Right hand side shows the same data retrieval in Goo.

development today often includes writing SPARQL queries. In many cases, software developers are unaware of what queries the SPARQL server is optimized for and what queries they should use. Furthermore, many software-development teams do not yet have SPARQL experts, which makes relying on triplestores as components in large software systems problematic. Indeed, even in our team we have experienced this problem as we were redesigning the BioPortal backend to use a triplestore and not all of our developers were proficient in writing efficient SPARQL queries. The development of Goo enabled our team members to overcome that barrier using the technology that they were familiar with (Ruby). Goo is a completely general ORM for SPARQL and therefore other developers can use it in their projects defining their models that Goo will validate and relying on the SPARQL query optimization in Goo to access their triplestores.

In this paper we show one example of how using naive SPARQL can lead to unexpected bad performance (see Figure 3). Our preliminary study shows that the combination of n-ary relations in hand-written queries result in query time distributions that may not perform well. Figure 3 shows that a simple partioning strategy can help to alleviate this issue.

We are proponents of semantic technologies, and thus we were drawn to the idea of using ontologies to define our schemas, including the domains and the allowed values for attributes. However, we see two major advantages to a DSL like Goo. First, one of our goals was to make schema definitions easy to use for developers who are not familiar with ontologies or OWL, and using ontologies counteracts that goal. Second, ontologies traditionally entail new information and are not designed to validate constraints.

Goo enables software developers without significant experience with semantic technologies, to use SPARQL and RDF naturally and efficiently. The BioPortal developers have found it easy to start working with the Goo API; as a result, the transition to RDF and triple store technology within BioPortal was much faster and smoother than it would have been otherwise. Basic query definitions in Goo are intuitive because the combination of hashes and arrays to construct Goo query patterns resembles JSON structures and developers are very familiar with them. Knowing that following a few simple restrictions, such as only querying for attributes that are needed, freed them from having to worry about the performance of the data storage layer and allowed them just to focus on the business logic, which sped development time significantly.

Acknowledgments This work was supported by the National Center for Biomedical Ontology, under grant U54 HG004028 from the National Institutes of Health.

References

1. OWL2Java: A Java Code Generator for OWL. `http://www.incunabulum.de/projects/it/owl2java`
2. Kalyanpur, A., Jimenez, D.: Automatic Mapping of OWL Ontologies into Java. In: Proceedings of Software Engineering and Knowledge Engineering (2004)
3. Arto Bendiken, Gregg Kellogg, B.L., Borkum, M.: RDF.rb: Linked Data for Ruby. `http://rdf.rubyforge.org/`
4. Ben Lavender, A.B., Humfrey, N.J.: Spira: A Linked Data ORM for Ruby. `https://github.com/datagraph/spira`
5. Graham Higgins, P.C.: RDF Alchemy, `http://www.openvest.com/trac/wiki/RDFAlchemy`
6. Oren, E., Delbru, R., Gerke, S., Haller, A., Decker, S.: Activerdf: Object-oriented Semantic Web Programming. In: Proceedings of the 16th International Conference on World Wide Web. pp. 817–824. WWW '07, ACM, New York, NY, USA (2007), `http://doi.acm.org/10.1145/1242572.1242682`
7. Salvadores, M., Horridge, M., Alexander, P.R., Fergerson, R.W., Musen, M.A., Noy, N.F.: Using SPARQL to Query BioPortal Ontologies and Metadata. In: International Semantic Web Conference (2). pp. 180–195 (2012)
8. Vollel, M.: Jenabean: A library for persisting java beans to RDF. `http://code.google.com/p/jenabean/`
9. Whetzel, P.L., Noy, N.F., Shah, N.H., Alexander, P.R., Nyulas, C.I., Tudorache, T., Musen, M.A.: BioPortal: Enhanced functionality via new web services from the national center for biomedical ontology to access and use ontologies in software applications. Nucleic Acids Research (NAR) 39(Web Server issue), W541–5 (2011)