

**INTEGRATION OF HETEROGENEOUS DATA FOR PROTEIN ONTOLOGY  
DATABASE USING SEMANTIC WEB TECHNOLOGY**

by

Xiang Li

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment  
of the requirements for the degree of Master of Science in Bioinformatics and  
Computational Biology

Fall 2018

© 2018 Xiang Li  
All Rights Reserved

**INTEGRATION OF HETEROGENEOUS DATA FOR PROTEIN ONTOLOGY  
DATABASE USING SEMANTIC WEB TECHNOLOGY**

by

Xiang Li

Approved: \_\_\_\_\_  
Chuming Chen, Ph.D.  
Professor in charge of thesis on behalf of the Advisory Committee

Approved: \_\_\_\_\_  
Kathleen F. McCoy, Ph.D.  
Chair of the Department of Computer and Information Sciences

Approved: \_\_\_\_\_  
Levi T. Thompson, Ph.D.  
Dean of College of Engineering

Approved: \_\_\_\_\_  
Doug Doren, Ph.D.  
Interim Vice Provost for Graduate and Professional Education

## **ACKNOWLEDGMENTS**

I wish to thank my adviser, Dr. Chuming Chen, and my committee members Dr. Hongzhan Huang, and Dr. Li Liao for their continuous advice, guidance, and academic support during the past year. I must also thank my professional friend and colleague, Mr. Wenbo Zhao, who has supported and helped me throughout my graduate education.

This thesis is dedicated to my parents Feng Li and Hua Li for their unconditional love and support, also to my wife Yaling Shi for her encouragement and inspiration.

## TABLE OF CONTENTS

LIST OF TABLES .....	vi
LIST OF FIGURES .....	vii
ABSTRACT .....	ix

### Chapter

1	INTRODUCTION .....	1
1.1	Semantic Web.....	1
1.1.1	RDF .....	3
1.1.2	RDF SCHEMA (RDFS) .....	5
1.1.3	OWL .....	6
1.1.4	SPARQL.....	6
1.2	Protein Ontology Database.....	7
1.2.1	Virtuoso SPARQL server .....	8
1.3	Application Programming Interface (API) .....	9
1.3.1	Python Django Framework.....	10
1.4	Outline of Thesis Work .....	13
2	SYSTEM DESIGN AND ARCHITECTURE.....	15
2.1	Design Rationale .....	15
2.2	System Architecture .....	18
2.3	Data Integration .....	19
3	VIRTUOSO/SPARQL BASED SEARCH ENGINE.....	25
3.1	PRO Search Website .....	25
3.2	SPARQL Syntax.....	28
3.3	SPARQL Query Library for PRO .....	30
3.4	Performance Evaluation .....	32

4	RESTFUL API .....	35
4.1	API Design .....	35
4.2	API Implementation .....	37
4.3	Use Cases.....	45
5	DISCUSSION AND FUTURE WORK .....	51
6	CONCLUSION .....	53
	REFERENCES .....	54
Appendix		
A	COMPARSION OF PERFORMANCE IN NEW/OLD DATABASE .....	58
B	CATEGORY OF PRO API .....	60

## LIST OF TABLES

Table 2.1	.....	24
-----------	-------	----

## LIST OF FIGURES

Figure 1.1 The components of the Semantic Web .....	2
Figure 1.2 RDF triple graph .....	4
Figure 1.3 An example RDF graph for a PRO term of Protein Ontology .....	4
Figure 1.4 The example of RDF Schema .....	5
Figure 1.5 The structure of Protein Ontology .....	8
Figure 1.6 The Flow Request and Response in Django Framework .....	11
Figure 1.7 An example RESTful API.....	13
Figure 2.1 LAV and GAV .....	15
Figure 2.2 Comparison between different methods of data integration .....	16
Figure 2.3 The architecture of old PRO database.....	18
Figure 2.4 The architecture of new PRO database .....	19
Figure 2.5 Sample data for PRO_extra RDF graph.....	20
Figure 2.6 Example of converting data into RDF in Turtle format.....	22
Figure 2.7 Example RDF triples in PRO_extra graph in Turtle format .....	23
Figure 3.1 PRO search website .....	26
Figure 3.2 Quick Links for PRO text search website.....	27
Figure 3.3 PRO main search interface.....	28
Figure 3.4 Basic structure of SPARQL query .....	29
Figure 3.5 Example SPARQL query for PRO field search .....	31
Figure 3.6 Average response time of 10 search queries .....	33

Figure 4.1 PRO RESTful APIs and Swagger UI interface.....	35
Figure 4.2 The data structure definition of PRO term.....	36
Figure 4.3 Python class for search parameters .....	37
Figure 4.4 Python class for retrieval parameters .....	39
Figure 4.5 Flow chart of function in Views .....	40
Figure 4.6 The layout of function definition in Views .....	40
Figure 4.7 Function for constructing SPARQL query dynamically .....	42
Figure 4.8 URL patterns defined in the Controller.....	43
Figure 4.9 An example JSON response from PRO RESTful APIs .....	44
Figure 4.10 An example XML response from PRO RESTful APIs.....	45
Figure 4.11 Inputs to “Search PRO terms” API. ....	46
Figure 4.12 A list of PRO_terms returned by “Search PRO terms” API. ....	47
Figure 4.13 Inputs to “Search organism-gene” API. ....	47
Figure 4.14 A list of organism specific PRO terms as returned by “Search organism-gene” API. ....	48
Figure 4.15 Inputs to “Search decedents” API. ....	49
Figure 4.16 A list of organism specific PRO terms as returned by “Search decedents” API. ....	49
Figure 4.17 Inputs to “Get PAF annotation” API.....	49
Figure 4.18 Annotations returned by “Get PAF annotation” API.....	50
Figure 4.19. Python script executes the search of a PRO term by its id. ....	50



## **ABSTRACT**

As the volume and diversity of data and the desire to share them increase, we inevitably encounter the problem of combining heterogeneous data generated from many different but related sources and the problem of providing users with a unified view of this combined data set. Data integration systems facilitate information access and reuse by providing a common access point and a more complete view of the available information. A widely adopted system, Semantic Web, provides the requisite technologies to make such integration possible: 1) an abstract model for the relational graphs: RDF; 2) a query language adapted for the relational graphs: SPARQL; and 3) various technologies to characterize the relationships and categorize resources: RDFS, OWL etc.

PRO databases draw on data sources that provide orthology, annotation, and mapping information, as well as sequence-related data, including amino acid and splice variants and multiple sequence alignments. The PRO website is currently hosted in two places: University of Delaware for entry page and visualization, and Georgetown University for text search and browse. The dual-site structure requires that data files be duplicated and overlapped, thus creating website maintenance issue. To streamline the update process and to remove redundancy, we explored simplifying the data integration for the PRO database using Semantic Web technology. In this process, the heterogeneous data was converted into RDF triples and integrated into a Virtuoso RDF triple store. Furthermore, a Virtuoso/SPARQL based search engine for the full-scale text search and hierarchy browsing for PRO website was developed.

Tests reveal that we achieved similar performance as compared to the Apache Lucene based search engine currently being used. We also developed RESTful APIs for programmatic access to the PRO database using Open API specification and Django REST framework.

In conclusion, the semantic web technologies such as RDF and SPARQL etc. are suitable for data integration. Heterogeneous data in the PRO database are structured and simplified by using RDF triples so that search efficiency can be improved. In addition, the thesis showed the design and implementation of the RESTful APIs in detail along with application examples. The thesis aims to provide a clear description of the heterogeneous data integration process and API design and implementation process that can be used as a reference in the field of Bioinformatics.

## **Chapter 1**

### **INTRODUCTION**

Recent rapid development of biotechnology leads to an explosive growth of bioinformatics data. However, very often the data are stored in different formats and in different databases. The information returned from a single database is usually not complete. Therefore, integration of heterogeneous data from different sources is necessary. By converting and combining them into a unified semantic model such that computer can process and understand to facilitate human-computer and computer-computer interactions is called the semantic data integration [1].

#### **1.1 Semantic Web**

Sir Tim Berners-Lee proposed the concept of “Semantic Web” in Scientific American article in 2001 [2]. In that article, he envisioned it as an evolution from the World Wide Web to the Semantic Web. The resources on the web are the documents, mainly for human consumption. In terms of current technology, it is impossible for computer to understand the meaning of the contents in the web documents accurately due to the complexity of unstructured data in the documents and lack of background knowledge. In 2006, Tim Berners-Lee coined the linked data [3]. It is the principles of publishing structured data so that they can be interlinked and more suitable for the semantic queries [4].

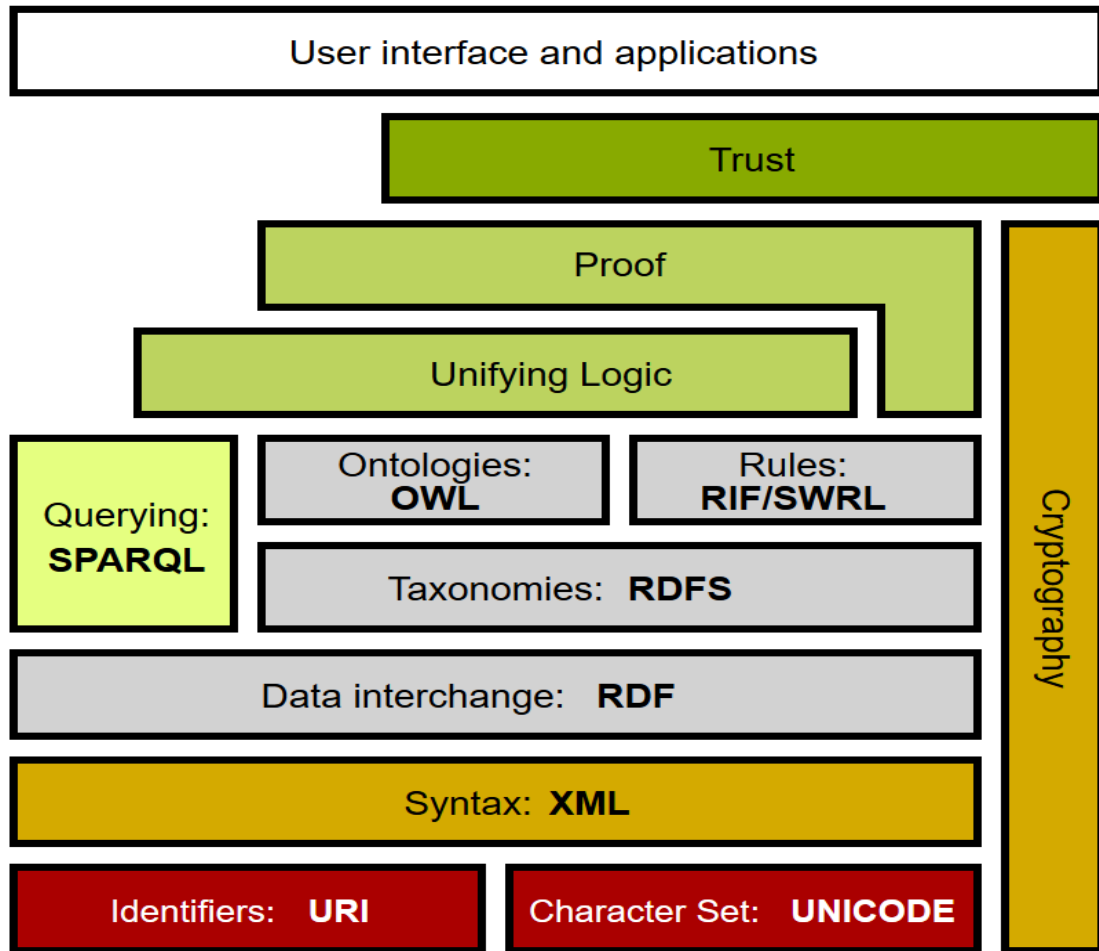


Figure 1.1 The components of the Semantic Web [5]

Figure 1.1 shows the major components of the Semantic Web. The Syntax layer is XML which is the markup language for structured data. The proof component is to determine if an answer found on the Semantic Web is correct, and support proof generation, exchange, and validation. The URI is used to uniquely identify Semantic Web resources and the UNICODE is the default character set used on the Semantic

Web. In addition, there are some important components such as RDF, RDFS and SPARQL that will be introduced in the following sections.

### **1.1.1 RDF**

The Resource Description Framework (RDF) is a framework for expressing information about resources [6]. The resources can be almost anything, including documents, people, abstract objects. RDF is designed to describe a model to represent the web resources. RDF uses the triples in the form of subject–predicate–object to make statements about resources. The Subject is a URI identifying the resource. The Predicate is a URI indicates the relationship between Subject and Object, and the Object is a literal value or URI of another resource related to the Subject. It can be represented as a directed-labeled graph (See Figure 1.2). Due to its data-centric architecture built upon a standardized model, RDF can exchange information between different applications and provide a standard-compliant way for data publication and interchange. These make RDF the most suitable framework for data integration. This is one of the reasons we use RDF model in this research project. Another reason is that RDF provides linked data framework, where heterogeneous data (structured, semi-structured and unstructured) can be expressed, stored and accessed in the same manner. This is made possible because the data structure is expressed through the links within the data itself instead of being constrained to a structure imposed by the relational database schema. As changes in the data structure occur, they are reflected in the RDF

database through changes in the links within the data. Interlinked datasets enable cross-dataset queries to be performed using SPARQL.

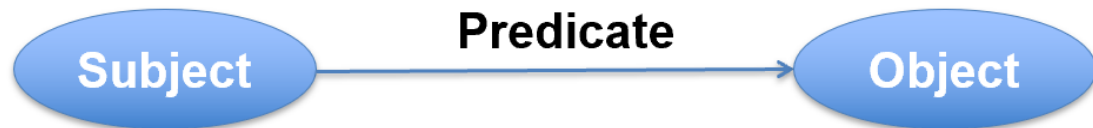


Figure 1.2 RDF triple graph

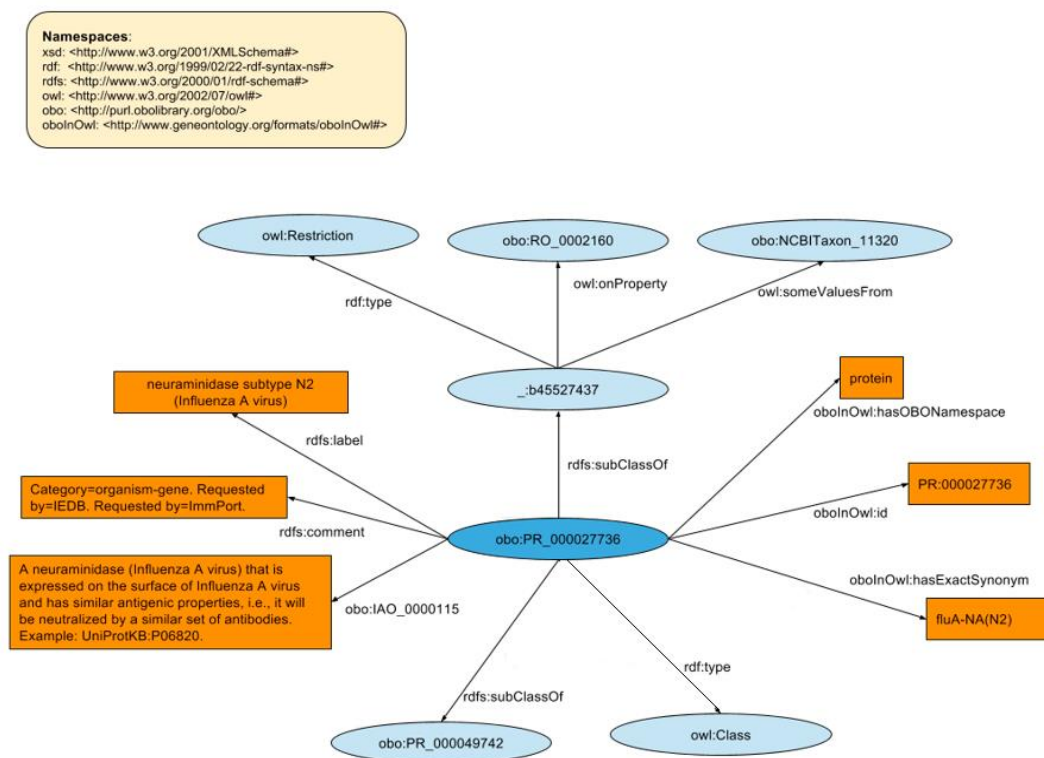


Figure 1.3 An example RDF graph for a PRO term of Protein Ontology

Figure 1.3 shows an example RDF graph for a PRO term of Protein Ontology. An RDF graph consists of nodes and edges. Nodes represent entities/resources (light blue nodes), attributes (orange nodes), and edges represent the relationship between entities and entities and the relationship between entities and attributes.

### 1.1.2 RDF SCHEMA (RDFS)

Resource Description Framework Schema (RDFS) is an extension vocabulary from the basic RDF vocabularies. When the RDF was proposed, people find that there are not enough classes and properties to describe the entities and attributes in the real world. Therefore, W3C published the RDFS as W3C recommendation in 2004 that is a supplement for RDF.

<pre> &lt;?xml version="1.0" encoding="utf-8" ?&gt; &lt;rdf:RDF    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"   xmlns:owl="http://www.w3.org/2002/07/owl#"    xmlns:oboInOwl="http://www.geneontology.org/formats/oboInOwl#"   xmlns:obo="http://purl.obolibrary.org/obo/" &gt;   &lt;rdf:Description     rdf:about="http://purl.obolibrary.org/obo/PR_000027736"   &gt;     &lt;rdf:type       rdf:resource="http://www.w3.org/2002/07/owl#Class" /&gt;     &lt;rdfs:label       rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;neuraminidase subtype N2 (Influenza A virus)&lt;/rdfs:label&gt;     &lt;rdfs:comment       rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;Category=organism-gene. Requested by=IEDB. Requested by=ImmPort.&lt;/rdfs:comment&gt;     &lt;rdfs:subClassOf       rdf:resource="http://purl.obolibrary.org/obo/PR_000049742" /&gt; </pre>	<pre> &lt;rdfs:subClassOf rdf:nodeID="b45527437" /&gt; &lt;oboInOwl:hasExactSynonym   rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;fluA-N A(N2)&lt;/oboInOwl:hasExactSynonym&gt;   &lt;oboInOwl:hasOBONamespace     rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;protein&lt;/oboInOwl:hasOBONamespace&gt;     &lt;oboInOwl:id       rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;PR:00 0027736&lt;/oboInOwl:id&gt;     &lt;obo:IAO_0000115       rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;A neuraminidase (Influenza A virus) that is expressed on the surface of Influenza A virus and has similar antigenic properties, i.e., it will be neutralized by a similar set of antibodies. Example: UniProtKB:P06820.&lt;/obo:IAO_0000115&gt;     &lt;/rdf:Description&gt;     &lt;rdf:Description rdf:nodeID="b45527437"&gt;       &lt;rdf:type         rdf:resource="http://www.w3.org/2002/07/owl#Restriction" /&gt;       &lt;owl:onProperty         rdf:resource="http://purl.obolibrary.org/obo/RO_0002160" /&gt;       &lt;owl:someValuesFrom         rdf:resource="http://purl.obolibrary.org/obo/NCBITaxon_11320" /&gt;     &lt;/rdf:Description&gt;   &lt;/rdf:RDF&gt; </pre>
--	---

Figure 1.4 The example of RDF Schema

As shown in Figure 1.4, RDFS consists of a set of classes with certain properties using the RDF extensible knowledge representation data model. In addition, RDFS defines properties rely on the classes of sources to which they apply, and classes depends on the properties they may have. Through mutual constraints, the RDFS can build a more rich and accurate vocabularies for describing resources in RDF.

### **1.1.3 OWL**

OWL is the Web Ontology Language. As mentioned above, RDFS is essentially an extension of the RDF vocabulary. However, people found that the expressivity of RDFS was still quite limited, so OWL was proposed [7]. OWL can also be seen as an extension of RDFS that provides additional mechanism for defining expressive and complex relationships on the Semantic Web. OWL provides a family of knowledge representation languages for developing ontologies on the Semantic Web.

### **1.1.4 SPARQL**

After data are linked and stored on the Semantic Web, user needs a language and tool to query, retrieval and manipulate them. In 2008, SPARQL 1.0 was published and recommended by W3C [8]. SPARQL stands for SPARQL Protocol and RDF Query Language [9]. SPARQL is a RDF query language. SPARQL can retrieve and



manipulate data stored in RDF format [10]. It allows users to write queries against what can loosely be called "key-value" data and the entire graph is a set of "subject-predicate-object" triples which is different from relational database. In terms of SQL relational database, the RDF can be seen as the table with three columns. Compared to noSQL database, the object column in RDF is heterogeneous. For instance, each predicate can have many different entries and even can return a collection. SPARQL provides a full set of analytic query operations such as JOIN, SORT, AGGREGATE where schema is intrinsically part of the data rather than requires a separate schema definition [11].

## **1.2 Protein Ontology Database**

The biological ontology is used to describe different conceptual frameworks that guide the collection, organization and publication of biological data [12].

The PRotein Ontology (PRO) [13], the reference ontology for protein entities in the OBO (Open Biological and Biomedical Ontologies) Foundry, represents protein families, multiple protein forms (proteoforms) arising from single genes, and protein complexes (Figure 1.5). In addition to the main ontology file itself (in OBO or OWL format), PRO databases draw on data sources that provide orthology, annotation, and mapping information, as well as sequence-related data, including amino acid and splice variants and multiple sequence alignments. These contribute to the underlying data resources to support PRO website by providing PRO entry view, batch retrieval and search functionalities. The PRO website is currently hosted in two places:

University of Delaware for entry page and visualization, and Georgetown University for text search and browse. The dual-site structure requires that data files be duplicated and overlapped, thus creating website maintenance issue.

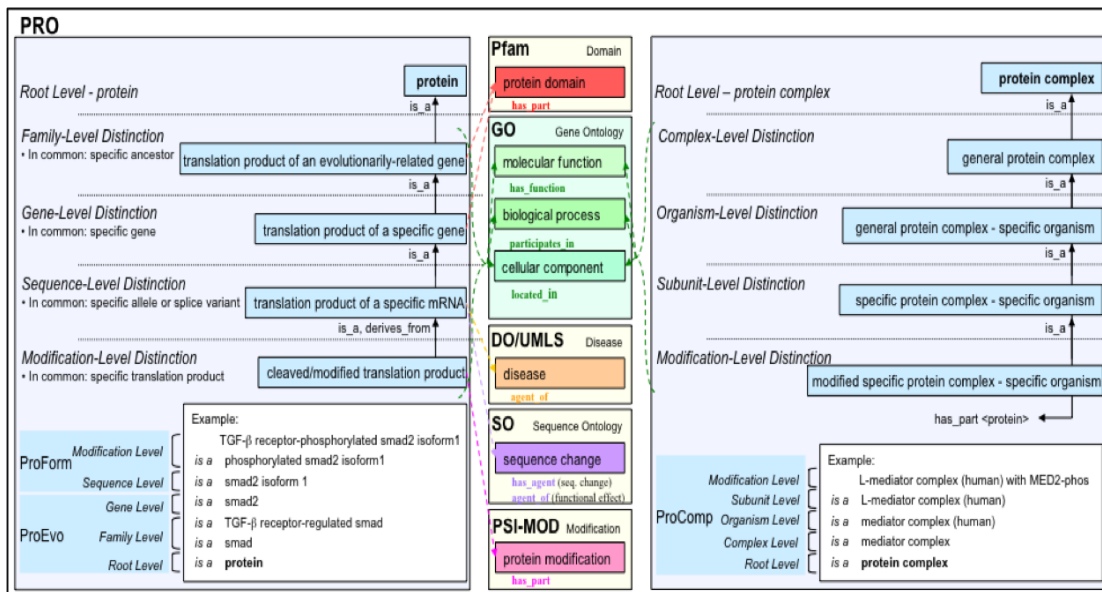


Figure 1.5 The structure of Protein Ontology [14]

### 1.2.1 Virtuoso SPARQL server

Virtuoso server is a special purpose-built and optimized database for the storage and retrieval of triples via semantic query language [15]. Virtuoso puts triples in a single table with the graph URI as a key (Quad Store). In addition to queries, triples can be imported/exported using RDF and other formats. Virtuoso server is also a SPARQL Service Endpoint. It supports SPARQL 1.1 and provides federated SPARQL query-processing for RDF data available on the Web. The latest version is

7.2. Virtuoso can reduce the cost of bringing data from different sources and make it more convenient to query and retrieval.

### 1.3 Application Programming Interface (API)

In computer science, an application-programming interface (API) is a set of subroutine definitions, communication protocols, and tools for building software. A good API makes it easier to develop a computer program by providing all the building blocks (Wikipedia). Representational State Transfer (REST) is an architectural style that defines a set of constraints to be used for creating web services [16]. We choose RESTful API to provide programmatic access to PRO RDF database. Because RESTful API has the following advantages [17].

1. *Client-Server Mode*. RESTful API separates user interface concern from the data storage concern. It supports developing portable user interface for multiple platforms and increasing the scalability of service because the components of service is reduced.
2. *Stateless*. The request sends from the client must include all the information so that the client keeps the complete session state. It also makes debugging easy for developer.
3. *Cacheable*. The response for a request can be implicitly or explicitly labeled as cacheable or non-cacheable. If it is cacheable, the service can cache it and reuse it. This feature can reduce the number of interactive connections and improve system response speed.

4. *Layered system.* APIs play the middle layer between the server and the client to respond to the client's request. The client does not need to care about anything other than the component that it interacts with. This not only improves the scalability of the system but also simplifies the complexity of the system. Because of those nice features, many large technology companies, such as Google, Amazon and Twitter, have widely used RESTful APIs. Most public bioinformatics resource and databases provides API services. For example, EUtilities API system of NCBI and the PSAMM API of Computational Molecular Ecology Lab at the University of Rhode Island [18]. These APIs can help developers getting results more conveniently and apply them into their project or research without rebuilding a big local database. Since the Protein Ontology database has been built for bioinformatics researchers, we also want to design our own APIs for PRO users and hope to make them jobs more productive.

### **1.3.1 Python Django Framework**

We used Python Django REST framework to develop RESTful API for our Virtuoso/SPARQL search engine-based PRO RDF database. In addition, we used several technologies to support our API design. The Open API specification, which is also previously called Swagger specification, is an API description format for REST APIs. It specifies the machine-readable interfaces and supports describing, producing, consuming, and visualizing of RESTful API [19]. We used Swagger Editor, a

browser-based editor to write the Open API specifications. We also used Swagger UI to render Open API specifications as interactive API documentation [20].

There are many advantages of Django framework. After more than a decade of development and improvement, Django has a wide range of practical use cases and comprehensive online documentation. Developers can search online documentation for solutions when they encounter problems. In addition, Django comes with a lot of tools and functionality common to many applications. It is also very convenient to manage the information from a database. There are four important parts of API framework: Models, Views, Controllers, Template. Theoretically, Django is an MVC framework, but part of the controller that accepts user input is handled by the framework itself, so Django is more like Models, Templates, and Views, also called MTV mode [21] (See Figure 1.6):

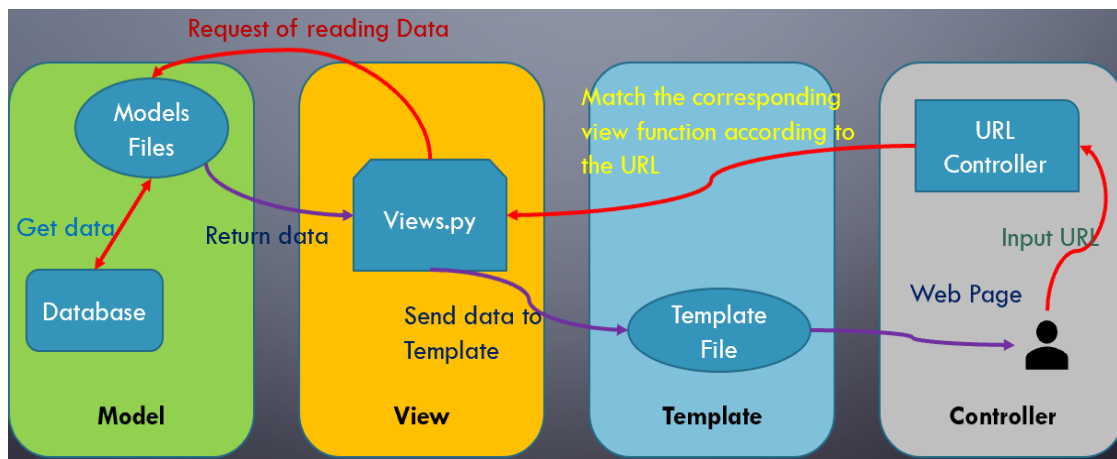


Figure 1.6 The Flow Request and Response in Django Framework

M, stands for Model, is the data access layer. This layer handles all transactions related to the data: how to access, how to verify validity, what behaviors to include, and deal with the relationship between data.

T, stands for Template, is the presentation layer. This layer handles performance-related decisions: How to display in a page or other type of document.

V, stands for View, is the business logic layer. This layer contains the relevant logic for accessing the model and getting the appropriate template. You can think of it as a bridge between the model and the template.

The API call is the process that the controller receives the request from the user and passes it to the view to get the data from model and generate the response formatted according to the template and returns to the user.

Request:

Method: GET

Request URL

```
https://research.bioinformatics.udel.edu/PRO_API/V1/pros?searchField=AllFields&searchValue=12345&showPROName=true&showPROTermDefinition=true&showCategory=true&showParent=true&showAnnotation=true&showAnyRelationship=true&showChild=true&showEcoCycID=false&showGeneName=true&showHGNCID=false&showMGIID=false&showOrthoIsoform=false&showOrthoModifiedForm=false&showPANTHERID=false&showPIRSFID=false&showPMID=false&showReactomeID=false&showUniProtKBID=false&Offset=0&Limit=50
```

Response body

```
[
  {
    "category": "gene",
    "name": "protocadherin-12",
    "parent": "PR:000000001",
    "geneName": "",
    "termDef": "A protein that is a translation product of the human PCDH12 gene or a 1:1 ortholog thereof.",
    "annotation": [],
    "anyRelationship": ""
  },
  {
    "category": "modification",
    "name": "C-prM",
    "parent": "PR:000036818",
    "geneName": "",
    "termDef": "A dengue virus genome polyprotein proteolytic cleavage product that is derived from dengue virus genome polyprotein, glycosylated form and that consists only of the C (capsid) protein and the prM (pre-membrane) structural proteins of the dengue virus genome polyprotein. It has a single N-linked glycosylation site at position Asn-69. UniProtKB:P29990, 1-280, Asn-183, MOD:00160.",
    "annotation": [],
    "anyRelationship": "PR:000036818 ! dengue virus genome polyprotein proteolytic cleavage product;PR:000036818 ! dengue virus genome polyprotein proteolytic cleavage product;derives from PR:000036817 ! dengue virus genome polyprotein, N-glycosylated form;has part MOD:00160 ! N4-glycosyl-L-asparagine;has part PR:000036819 ! capsid protein C;has part PR:000036837 ! prM;lacks part PR:000036838 ! NS12345;derives from PR:000036817 ! dengue virus genome polyprotein, N-glycosylated form;has part MOD:00160 ! N4-glycosyl-L-asparagine;has part PR:000036819 ! capsid protein C;has part PR:000036837 ! prM;lacks part PR:000036838 ! NS12345;"
  },
]
```

Figure 1.7 An example RESTful API

As shown in Figure 1.7, user inputs a Request URL, the API service returns the Response. By adding different renders and information in the request header, the response can be rendered in JSON or XML format.

## 1.4 Outline of Thesis Work

The first step is to convert the source data power the current PRO database into RDF triples. We study the content and organization of each data source, identified and extracted relevant information, converted them into RDF triples. Python script is developed to extract unstructured data from original source files and convert them into

RDF triples. Meanwhile, the duplicate information in those source files are identified and removed. The second step is to load those RDF triples into Virtuoso triple store as three named graphs. The third step is to build a SPARQL query library and search engine that supports quick link search and Boolean clause-based search of current PRO search website. Finally, accuracy and performance tests are done to evaluate the Virtuoso/SPARQL based search engine.

To facilitate programmatic access the PRO RDF database and SPARQL search engine, we design the PRO RESTful APIs based on Open API specification and using Swagger editor. We then implement the APIs using Django REST framework.



## Chapter 2

### SYSTEM DESIGN AND ARCHITECTURE

The data heterogeneity increases the system complexity and hinders its performance. We therefore need new methodology and model to do data integration.

#### 2.1 Design Rationale

Lenzerini proposed a logical framework for data integration systems from a theoretical perspective based on the notion of global schema, where the goal of a data integration system is to provide the users with a homogeneous view of the data across different sources [22]. In this theoretical model, data integration can be characterized into two approaches: LAV (Local-As-View) versus GAV (Global-As-View).

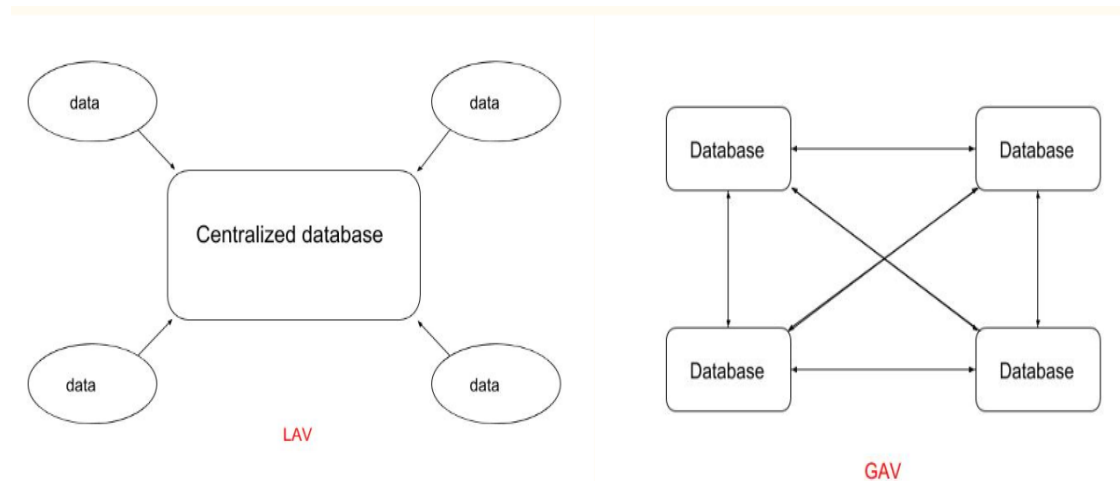


Figure 2.1 LAV and GAV

As shown in Figure 2.1, the LAV approach is the most effective approach when the global schema is stable in the data integration system. A typical example of this approach is data warehouse. The data warehouse approach puts data sources into a centralized location with a global data schema and an indexing system for fast data retrieval. The GAV approach is the most effective approach when the set of sources are stable in the data integration system. The example of this approach is federated database. The federated database approach does not require a centralized database. It maintains a common data model and relies on a schema mapping to translate heterogeneous database schema into the target schema for integration. Therefore, it is modular, flexible, and scalable. This project uses the LAV as the model to integrate heterogeneous data for PRO database.

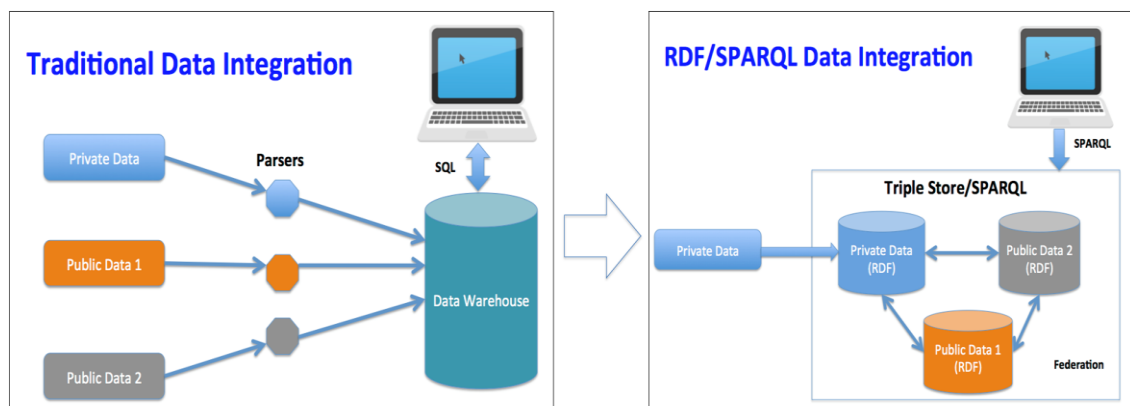


Figure 2.2 Comparison between different methods of data integration

The Figure 2.2 is the description of different integration methods. The traditional way of integration is GAV model which using the extra parsers to analysis

and form the data extracted from different databases. The typical example is centralized data warehouse that combines data from different sources and user can query these data using SQL. In contrast, the method used in this thesis project is distributed. There is no centralization in this model and user can use the federated SPARQL to query them.

The data heterogeneity is well known in the current PRO database as shown in Figure 2.3. The PRO database consists of four different kinds of databases: SQL-lite, Oracle, Virtuoso in the University of Delaware and Apache Lucene Search Engine in the Georgetown University. Each database stores part of the data, which may have already been stored in other databases and in different formats. At the same time, the returned result is incomplete because they use different query languages. It is very complicated for user and the requirements for maintenance personnel are also increased since they need to maintenance four different databases at the same time. It is also not easy to update if there is a wrong data need to modify because there are duplicated in many databases. Therefore, it is necessary to integrate these data to improve the efficiency of query and update in the PRO database and website.

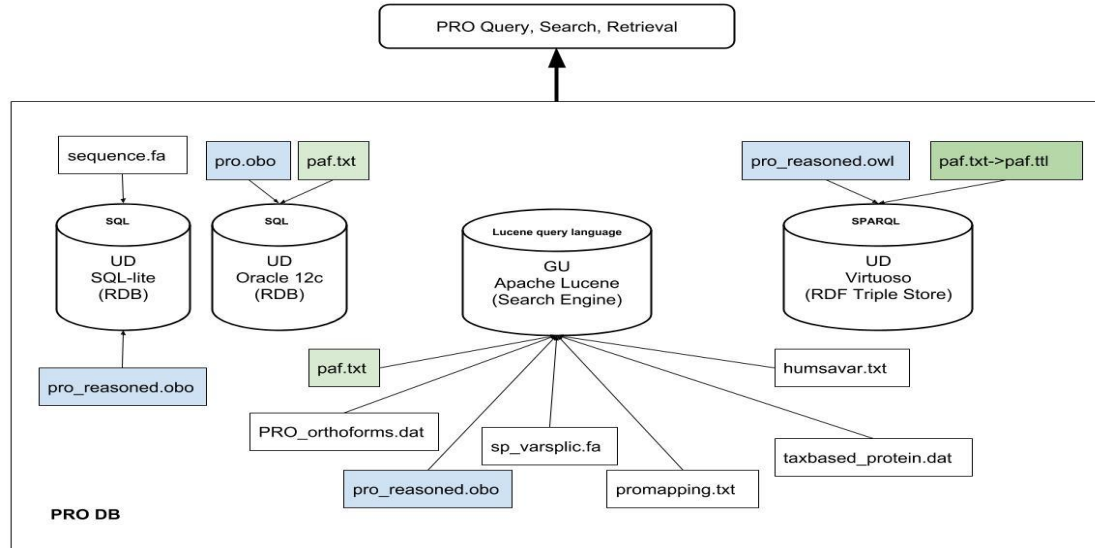


Figure 2.3 The architecture of old PRO database

## 2.2 System Architecture

As shown in Figure 2.3, other than the Protein Ontology, which is the core of the PRO database, there are additional data such as PRO\_orthoforms, PAF (PRO annotation file), PRO\_mapping, Human protein variants, Splice variants, Protein sequences, MSA (Multiple Sequence Alignment) etc. They are used to power the PRO website to provide PRO entry view, batch retrieval and search functionalities. We can also see from Figure 2.3, not only different database technologies and query languages are used, they are also been hosted on different sites: University of Delaware (UD) and Georgetown University (GU). In addition, the same source data is repeatedly used in

different databases. This creates potential synchronization issue during PRO database update cycle. This thesis project explored simplifying the data integration for PRO database using Semantic Web technologies as shown in Figure 2.4.

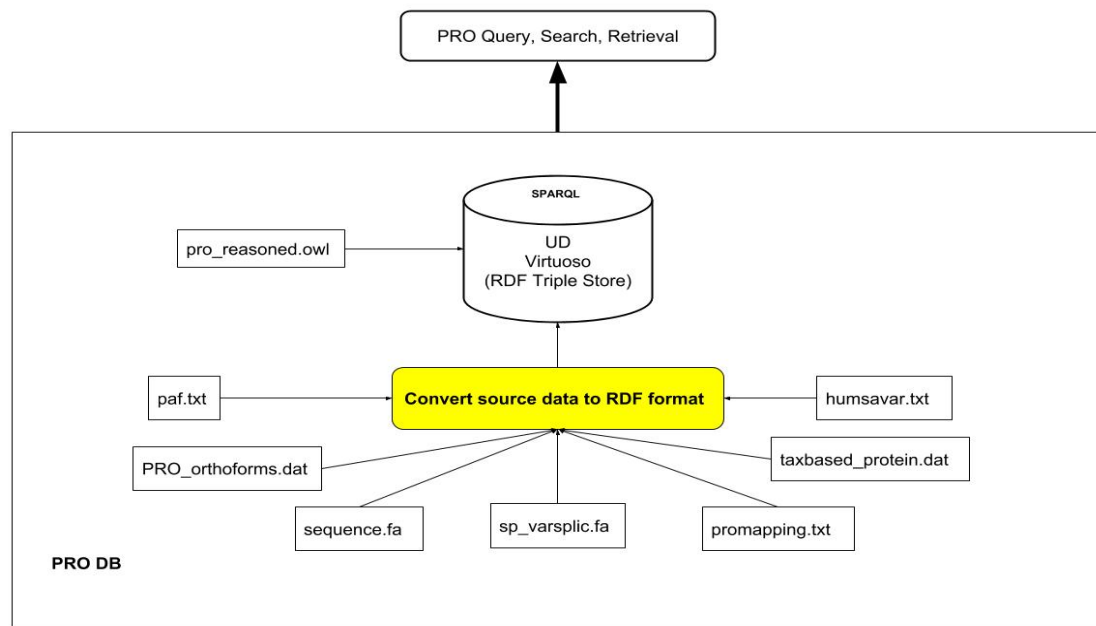


Figure 2.4 The architecture of new PRO database

## 2.3 Data Integration

To streamline the update process and to remove redundancy, we explored simplifying the data integration for the PRO database using Semantic Web technology. We studied the content and organization of each data source, identified and extracted

relevant information, converted them into RDF triples, and integrated them into a Virtuoso RDF triple store.

The most important step is to convert source data into RDF format. Most of the data for PRO database are from PRO OWL file and PAF file. They have been converted and stored in RDF format in the Virtuoso triple store. However, there are still some data spreaded across multiple files. We therefore extracted them mainly from ‘PRO\_orthoforms.dat’ and ‘pro\_reasoned.obo’ and converted them into RDF triples and stored in a new RDF graph called “PRO\_extra”.

```

resu = re.match(r'replaced_by: (PR:[A-Z0-9\-\[\]]*)',ln)
relation1 = re.match(r'is_a: (.*)',ln)
relation2 = re.match(r'intersection_of: (.*)',ln)
relation3 = re.match(r'relationship: (.*)',ln)

```

PRO\_orthoforms.dat

```

ortho-isoform PR:000039805 = PR:P55809-1 PR:Q9D0K2-1
ortho-isoform PR:000043072 = PR:Q9D0L7-1 PR:B1WBW4-1
ortho-isoform PR:000041606 = PR:Q5XIS2-1 PR:Q8BS40-1
ortho-gene PR:000001980 = PR:P70673 PR:Q61743 PR:Q2HX26 PR:Q14654
ortho-gene PR:000001097 = PR:Q6Y1R5 PR:Q96P68 PR:Q6IYF8
ortho-gene PR:000005626 = PR:Q8N3K9 PR:Q70KF4
ortho-gene PR:000005431 = PR:Q9QZD5 PR:P26374 PR:Q8LLD4
ortho-isoform PR:000042733 = PR:Q8K0X8-1 PR:Q99689-1
ortho-gene PR:000015070 = PR:A4ZYQ5 PR:P0C6A1 PR:Q6PXP3
ortho-gene PR:000022135 = PR:Q2G0F8 PR:P11875 PR:A5I0R5
ortho-gene PR:000015526 = PR:Q6P3D7 PR:Q8N5J4
ortho-modification PR:000000556 = PR:000026341* PR:000036464* PR:000036466*

```

PRO\_reasoned.obo

Figure 2.5 Sample data for PRO\_extra RDF graph

As shown in the Figure 2.5, there are seven different kinds of predicate. In the file ‘orthoform.dat’, they are ‘replaced by’, ‘is\_a’, ‘intersection\_of’ and ‘relationship’. Other three kinds of predicates, ‘ortho-gene’, ‘ortho-isoform’ and ‘ortho-modification’

are in the file 'PRO\_reasoned.obo'. They all have their own special definitions. For example, 'relationship' represents the related proteins and their relationships. 'Ortho-modification' means the proteins on the right of equal sign are modified by one or many methods from the original protein on the left.

Turtle (Terse RDF Triple Language) [23] is a format for expressing RDF triples in a compact textual form. Turtle provides a way to group three URIs to make a triple, and provides ways to abbreviate such information, for example by factoring out common portions of URIs.

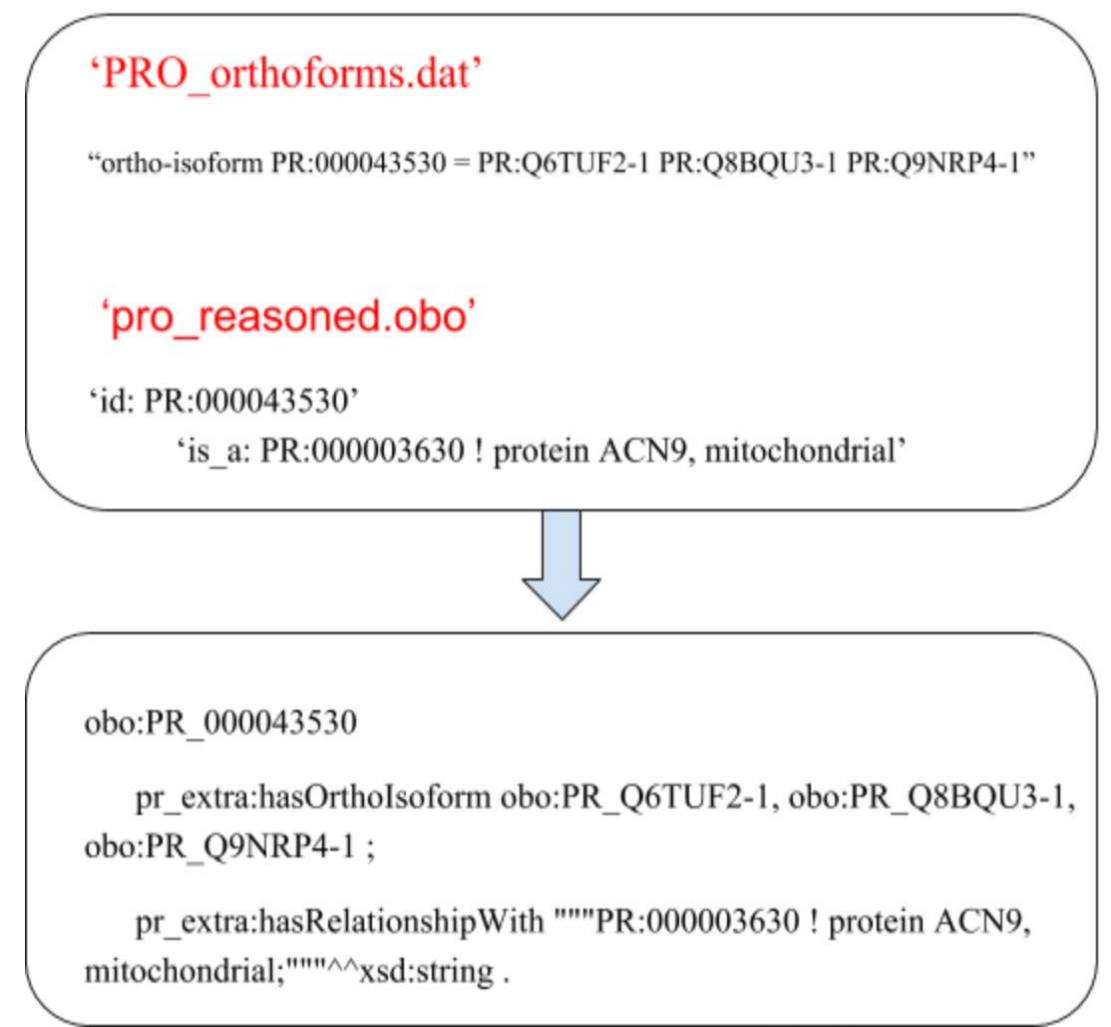


Figure 2.6 Example of converting data into RDF in Turtle format

Figure 2.6 describes the process of converting and combining data for PRO\_extra RDF graph. As the supplement to PRO RDF graph, classes in the PRO\_extra have been defined in the PRO RDF graph. Therefore, we reuse them. The data in “PRO\_orthoforms.dat” file has the relationship which is the “ortho-isoform”.



However, there is no predefined URI or predicate. Therefore, we created new predicate: “pr\_extra:hasOrthoIsoform”.

There are three principles in convert source data to RDF triple in Turtle format:

1. “prefix” is defined to represent namespaces and URIs sharing the same base.
2. Same subject can be referenced by a number of predicates. Therefore, a series of RDF triples can be written by a series of predicates and objects, separated by “;”, following a subject.
3. Objects are often repeated with the same subject and predicate. Therefore, a series of RDF triples can be written by a series of objects, separated by “,”.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix obo: <http://purl.obolibrary.org/obo/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix pr_extra: <https://proconsortium.org/pr_extra#> .

pr_extra:hasOrthoIsoform
  rdf:type owl:ObjectProperty ;
  rdfs:label "hasOrthoIsoform"^^xsd:string ;
  rdfs:domain owl:Class ;
  rdfs:range owl:Class ;
  obo:IAO_0000115 "Ortho-isoforms were true alternative isoforms in a common ancestor, and quite likely functionally
equivalent."^^xsd:string .

pr_extra:hasOrthoModifiedForm
  rdf:type owl:ObjectProperty ;
  rdfs:label "hasOrthoModifiedForm"^^xsd:string ;
  rdfs:domain owl:Class ;
  rdfs:range owl:Class ;
  obo:IAO_0000115 "Ortho-modified forms indicate the PTMs on ortho-isoforms that occur in equivalent residues.
"^^xsd:string .

pr_extra:isReplacedBy
  rdf:type owl:ObjectProperty ;
  rdfs:label "isReplacedBy"^^xsd:string ;
  rdfs:domain owl:Class ;
  rdfs:range owl:Class ;
  obo:IAO_0000115 "One PRO term is replaced by another PRO term."^^xsd:string .

pr_extra:hasRelationshipWith
  rdf:type owl:ObjectProperty ;
  rdfs:label "anyRelationship"^^xsd:string ;
  rdfs:domain owl:Class ;
  rdfs:range rdfs:Literal ;
  obo:IAO_0000115 "Any relationship between PRO term and ontology or database identifier"^^xsd:string .

obo:PR_Q8VF13
  pr_extra:hasRelationshipWith ""PR:000012115 ! olfactory receptor 1094;PR:000029032 ! Mus musculus protein;PR:
000012115 ! olfactory receptor 1094;only_in_taxon NCBITaxon:10090 ! Mus musculus;has_gene_template MGI:3030928 ! Olfr1094
(mouse);only_in_taxon NCBITaxon:10090 ! Mus musculus;"^^xsd:string .

obo:PR_014709
  pr_extra:hasRelationshipWith ""PR:000017742 ! zinc finger protein 197;PR:000029067 ! Homo sapiens protein;PR:
000017742 ! zinc finger protein 197;only_in_taxon NCBITaxon:9606 ! Homo sapiens;has_gene_template HGNC:12988 ! ZNF197
(human);only_in_taxon NCBITaxon:9606 ! Homo sapiens;"^^xsd:string .
```

Figure 2.7 Example RDF triples in PRO\_extra graph in Turtle format

The Figure 2.7 is the example of the integration result. “@prefix” is used to define the namespaces and base URI. There are some special properties uniquely defined for PRO\_extra. The conversion script is developed in Python. The decisive parameters in script are the regular expression rules. It is very convenient to add or modify those rules to extract data.

Table 2.1 Statistics of PRO RDF database (PRO version 56.0)

<b>Named Graph</b>	<b>Triples</b>	<b>Classes</b>	<b>Entities</b>	<b>Distinct Subjects</b>	<b>Properties</b>	<b>Distinct Objects</b>
<http://purl.obolibrary.org/obo/pr>	10,164,037	8	1,747,528	2034,478	45	2,911,165
<http://pir.georgetown.edu/pro/paf>	94,409	4	8,599	20,709	22	30,781
<https://proconsortium.org/pr_extra>	398,643	1	4	266,912	10	316,530

Table 2.1 shows the number of triples, classes, entities, distinct subjects, properties, and distinct objects for three named graphs stored in Virtuoso triple store for PRO RDF database (version 56.0).

## Chapter 3

### VIRTUOSO/SPARQL BASED SEARCH ENGINE

The search function of current PRO website is powered by Apache Lucene search engine. As a high performance, scalable information retrieval (IR) tool library [24], Lucene stores the index and data together, therefore Lucene can search the index rapidly and then the data can return directly without additional retrieval step. In our new architecture, we propose to build a Virtuoso/SPARQL based search engine by exploring the full-text search functionality of Virtuoso server to achieve the goals of data integration and high-performance information retrieval.

#### 3.1 PRO Search Website

We have two applications to demonstrate the usefulness of our SPARQL based search engine for PRO. One is the PRO text search website. Another one is the RESTful APIs that will be described in the Chapter 4.

Figure 3.1 shows the web interface for PRO text search website. The red box shows the quick links which includes some default filter conditions so that users can return search results directly for some specific queries. The blue box is main body of the query input. The search result is displayed as paginated table. The columns shown in the table can be further customized by the “Display Options”. The search interface was built with Perl CGI, HTML and Javascript. We re-used the front-end code and

replaced its underlying Apache Lucene based search engine with our Virtuoso/SPARQL based search engine.

215659 entries | 4314 pages | 50 / page | K << 1 2 3 4 5 >> K

Save Result As: TABLE

PRO ID	PRO Name	PRO Term Definition	Category	Parent
<a href="#">PR:X5M8U1-2</a>	receptor-type guanylate cyclase gcy-17 isoform b (worm)	A receptor-type guanylate cyclase gcy-17 (worm) that is a translation product of some mRNA giving rise to a protein with the amino acid sequence represented by UniProtKB: <a href="#">X5M8U1-2</a>	organism-sequence	<a href="#">PR:X5M8U1</a>
<a href="#">PR:X5M8U1-1</a>	receptor-type guanylate cyclase gcy-17 isoform a (worm)	A receptor-type guanylate cyclase gcy-17 (worm) that is a translation product of some mRNA giving rise to a protein with the amino acid sequence represented by UniProtKB: <a href="#">X5M8U1-1</a>	organism-sequence	<a href="#">PR:X5M8U1</a>
<a href="#">PR:X5M8U1</a>	receptor-type guanylate cyclase gcy-17 (worm)	A protein that is a translation product of the gcy-17 gene in worm	organism-gene	<a href="#">PR:000036194</a>
<a href="#">PR:X5M5N0-9</a>	serine/threonine-protein kinase WNK isoform h (worm)	A serine/threonine-protein kinase WNK (worm) that is a translation product of some mRNA giving rise to a protein with the amino acid sequence represented by UniProtKB: <a href="#">X5M5N0-9</a>	organism-sequence	<a href="#">PR:X5M5N0</a>
<a href="#">PR:X5M5N0-8</a>	serine/threonine-protein kinase WNK isoform g (worm)	A serine/threonine-protein kinase WNK (worm) that is a translation product of some mRNA giving rise to a protein with the amino acid sequence represented	organism-sequence	<a href="#">PR:X5M5N0</a>

Figure 3.1 PRO search website

The Quick Links (Figure 3.2) can be grouped into three sections: Modified forms, Terms related to disease and Other links. Modified protein forms are distinguished by their category descriptions. However, everyone has to have keyword “modification”. For “Terms related to disease”, we currently only have “Saliva biomarkers” that represents proteins having the database cross-reference of “SALO: AJ”. We also have other database cross-reference in “Other Links” section, such as EcoCyc, MGI, Panther, Reactome and UniProtKB. They have special filter conditions on ID. ‘Complex’, ‘Family Level’, and “Orthisoforms” are also included in this

section. These quick links allow user to get specific query results quickly. However, they can also be combined with the main query interface, which can be used with filter condition.

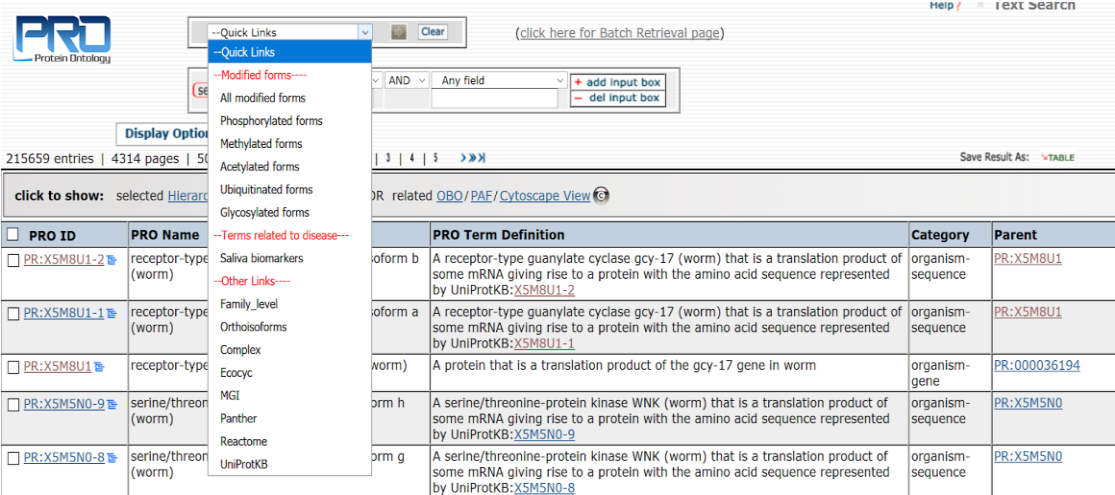


Figure 3.2 Quick Links for PRO text search website

Figure 3.3 shows the PRO main search interface. User can select the field and specify the search condition. User can also form Boolean query clause by clicking “add/del input box” button for different search fields. In addition to viewing the search result page by page, user can select a list of PRO IDs to show their Hierarchy or display their corresponding entries in OBO or PAF format. User can also view the list of PRO DIs in Cytoscape network view.

PIR Protein Information Resource

PRO Protein Ontology

Quick Links Clear (click here for Batch Retrieval page)

search Any field AND Any field add input box del input box

Display Options

215659 entries | 4314 pages | 50 / page | Save Result As: TABLE

click to show: selected Hierarchy selected OBO / PAF OR related OBO / PAF / Cytoscape View

PRO ID	PRO Name	PRO Term Definition	Category	Parent
PR:X5M8U1-2	receptor-type guanylate cyclase gcy-17 isoform b (worm)	A receptor-type guanylate cyclase gcy-17 (worm) that is a translation product of some mRNA giving rise to a protein with the amino acid sequence represented by UniProtKB:X5M8U1-2	organism-sequence	PR:X5M8U1

Figure 3.3 PRO main search interface

## 3.2 SPARQL Syntax

SPARQL has many advantages [25]. First, the level of standardization of implementations using RDF and SPARQL is much higher than SQL. It's possible to swap out one triple store for another easily. Second, SPARQL is expressive. It is much easier to model complex data in RDF than in SQL, and finally, it is easier to do things like LEFT JOINs (called OPTIONAL) in SPARQL.

```

# prefix declarations
PREFIX obo: <...>
..
# result clause
SELECT ...
# dataset definition
FROM <...>
FROM NAMED <...>
# query pattern
WHERE {...}
# query modifiers
GROUP BY ...
ORDER BY ...
LIMIT ...
OFFSET ...
VALUES ...

```

Figure 3.4 Basic structure of SPARQL query

SPARQL mainly consists of five components structurally. Except for the query modifiers, the rest of components are essential in SPARQL query. The prefix declaration is used to declare the namespaces which have included the stated entity and relationships. The result clause is for identifying what information to return from the query. Dataset definition is used to state what RDF graph(s) are being queried. It also can be seen as the range of search. The query pattern is the main body of SPARQL query. It shows the triple (graph) patterns people are searching in the query dataset and the matched result will be restricted by the result clause. The query

modifier has many functions such as slicing, ordering, and otherwise rearranging query results.

There are three RDF Terms in SPARQL syntax: the URI, the literal value and the variable. The URI and the literal are the basic type in the SPARQL and RDF. Variable represents any unknown thing in the triple pattern and the actual value can be projected in the result clause.

There are some special key words for different functions. The keyword “FILTER” is a restriction on solutions over the whole group in which the filter appears and helps people searching result more accurately. Another keyword is “OPTIONAL”, which allows additional patterns to extend the solution. Because there is no null value in the SPARQL, this keyword is designed for matching additional patterns that may extend the solution. It can allow information to be added to the solution where the information is available, but do not reject the solution because some parts of the query pattern do not match. We can use keyword “UNION” to combine graph patterns so that one of several alternative graph patterns may match. If more than one of the alternatives matches, all the possible pattern solutions are found. The last one is “GRAPH”. When querying a collection of graphs, the GRAPH keyword is used to match patterns against named graphs. The use of GRAPH changes the active graph for matching graph patterns within that part of the query. Outside the use of GRAPH, matching is done using the default graph.

### **3.3 SPARQL Query Library for PRO**

The SPARQL query for each field in this graphic interface needs to be built first. As shown in the example SPARQL query for PRO field search (Figure 3.5). The



red color codes show the triple patterns related to PRO term definition in Turtle format. The two cells on the right are its corresponding SPARQL queries. One for all PRO terms, another one for specified PRO terms.

The SPARQL Query for fields

Field	Example	Retrieval	Query
PRO_DEF_TXT (pro term definition)	obo:PR_000000001 obo:IAO_0000115 "An amino acid chain that is produced de novo by ribosome-mediated translation of a genetically-encoded mRNA."^^xsd:string ;	PREFIX obo: <http://purl.obolibrary.org/obo/> PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> PREFIX obolnOwl: <http://www.geneontology.org/formats/obolnOwl#> SELECT ?PRO_term ?PRO_name ?PRO_ID ?PRO_DEF FROM <http://purl.obolibrary.org/obo/pr> WHERE { ?PRO_term rdfs:label ?_PRO_name . ?PRO_term obolnOwl:id ?_PRO_ID . ?PRO_term obo:IAO_0000115 ?_PRO_DEF . BIND(str(?_PRO_name) as ?PRO_name) . BIND(str(?_PRO_ID) as ?PRO_ID) . BIND(str(?_PRO_DEF) as ?PRO_DEF) . }	PREFIX obo: <http://purl.obolibrary.org/obo/> PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> PREFIX obolnOwl: <http://www.geneontology.org/formats/obolnOwl#> SELECT ?PRO_term ?PRO_name ?PRO_ID ?PRO_DEF FROM <http://purl.obolibrary.org/obo/pr> WHERE { ?PRO_term rdfs:label ?_PRO_name . ?PRO_term obolnOwl:id ?_PRO_ID . ?PRO_term obo:IAO_0000115 ?_PRO_DEF . BIND(str(?_PRO_name) as ?PRO_name) . BIND(str(?_PRO_ID) as ?PRO_ID) . BIND(str(?_PRO_DEF) as ?PRO_DEF) . VALUES ?variable ("PR:X5M8U1") . FILTER(regex(?PRO_ID,?variable)) . }

Figure 3.5 Example SPARQL query for PRO field search

Because there are only three namespaces that are used in this example query, only three abbreviations are stated in PREFIX section. The line starting from SELECT is the result clause in the query. Depending on the field searched, user can select any related field they want to get the value from. The example is to query the definition of protein, so the basic requirement is the URI, name, ID and definition of a protein. After the dataset clause, three triple patterns are listed in the query patterns. In this

example, it is mostly basic patterns without any special keyword. The keyword “BIND” is just for getting only the literal portion and removing the URL from a variable. There are two steps involved. The first step is to create a variable to hold the value entered by user. This part is implemented by keyword “VALUES”, In SPARQL language, it is used for assigning a value to a variable. In Figure 3.5, the value “PR:X5M8U1” was given to the variable, “?variable”. The next step is to filter it with specified condition using the keyword “FILTER” introduced above. In the example, the combination of “FILTER” and “regex” is used to filter the value in the field of “PRO ID” with regular expression search. According to the field selected and value entered by the users, it will only work for one field. On PRO search interface, there are 26 fields. User can select multiple fields and construct Boolean clause query using operators “NOT”, “AND” and “OR”. To deal with this, a SPARQL query is split into different code snippets with respect to their corresponding fields and stored in a lookup table (Dictionary in Python). Based on the user input, different query snippets are combined dynamically to construct a complete SPARQL query.

### **3.4 Performance Evaluation**

We conducted performance evaluation using CURL command to record the query response times of 10 queries against Apache Lucene based and Virtuoso/SPARQL based search engines of PRO database. Each query was repeated 10 times for each search engine (Figure 3.6).

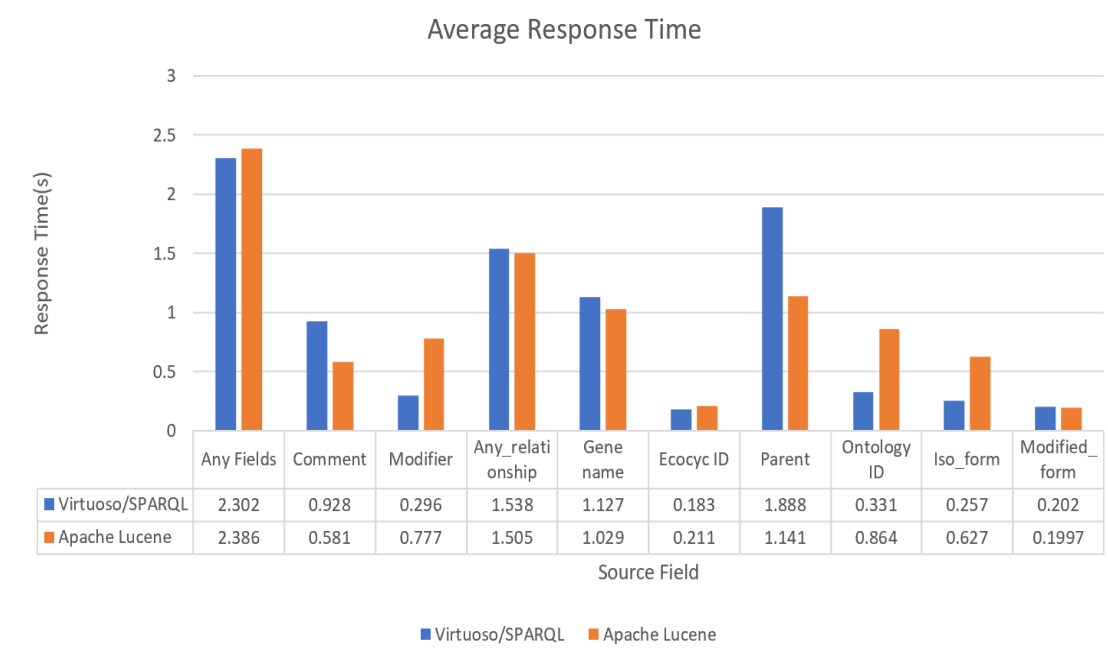


Figure 3.6 Average response time of 10 search queries

In Figure 3.6, there are 10 different fields. For example, “Ecocyc ID,” PRISF ID,” PMID” and “UniProtKB ID”. They have same data type and even similar triple pattern so there is only “Ecocyc ID” chosen as the sample in the performance test. When the property is the literal value, the efficiency of searching is similar and sometime better. In this test, these fields include the “Comment”, “Any relationship”, “Gene name” and “Ecocyc ID”. The annotation is the delegate of this kind of variable. In the chart, the field “Modifier” and “Ontology ID” were chosen as the property sample for Annotation. They all have a good performance and are better than the performance of Apache Lucene based search engine. In addition, the field “Iso-formed” and “Modified-form”, which belong to the PAF source, have a similar performance.

As shown in the chart, for fields “Parent”, the response time is increased significantly. Those are related to the complexity of graph patterns to match or whether we have alternative patterns to match. Overall, Virtuoso/SPARQL based search engine achieved comparable performance with respect to Apache Lucene based search engine. The details of performance evaluation can be found in Appendix A.

## Chapter 4

### RESTFUL API

#### 4.1 API Design

PRO API design (Figure 4.1) was motivated by PRO text search website and PRO manual curation guide website URI [26]. The API specification was designed using Swagger editor (<https://editor.swagger.io/>) based on Open API (formerly known as Swagger) Specification 3. Swagger UI was used to visualize and interact with the API's resources automatically generated from API specifications. The API is currently accessible at the referenced website [27].

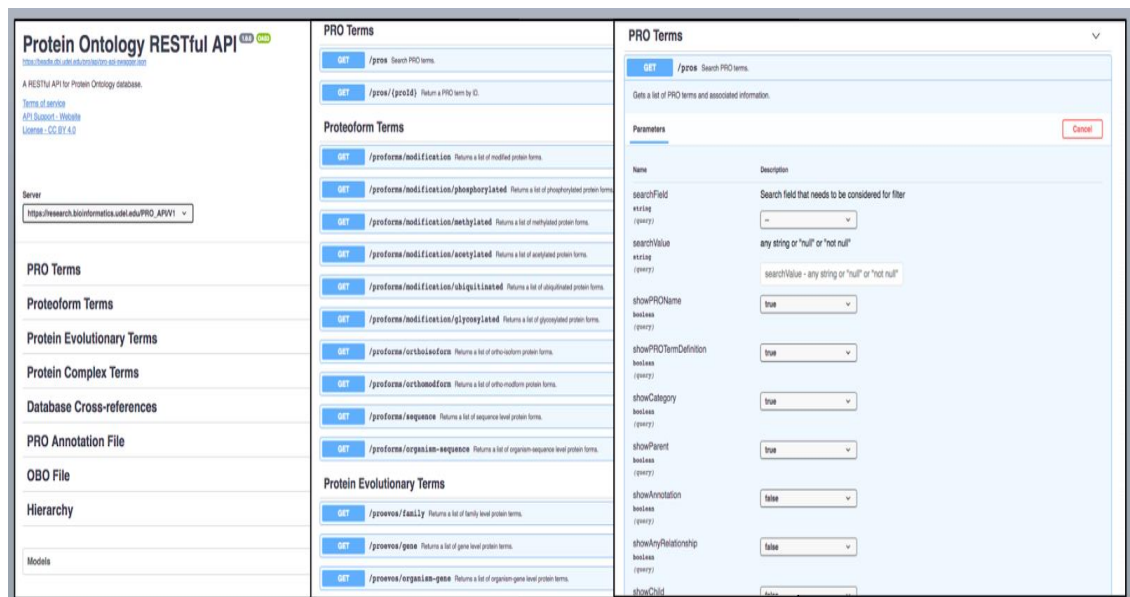


Figure 4.1 PRO RESTful APIs and Swagger UI interface

The PRO RESTful APIs include 8 API operation groups (PRO Terms, Proteform Terms, Protein Evolutionary Terms, Protein Complex Terms, Database Cross-references, PRO Annotation File, OBO File, Hierarchy) and 34 access paths. The description of each access path and its functionality can be found in Appendix B. The core model of PRO APIs is PRO term, which consists of a list of attributes associated with a given PRO term (Figure 4.2). For example, PRO ID, protein name, term definition, category etc. The PRO RESTful APIs only support GET method, which is the read-only operation. The API response can be in either JSON or XML format.

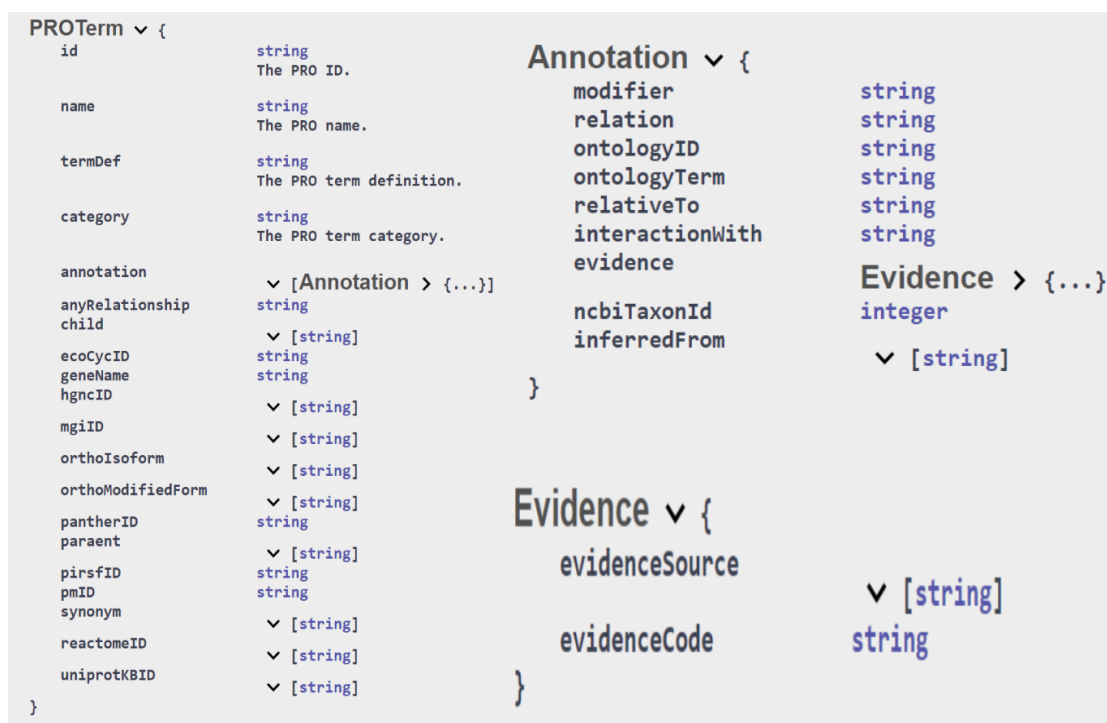


Figure 4.2 The data structure definition of PRO term

## 4.2 API Implementation

The PRO RESTful APIs are implemented in Python (version 2.7.15rc1) and Django-REST framework (version 1.11.15). The first step in API implementation is to build python classes for holding both the input parameters and query results. They are Models in terms of Django-REST framework. One reason we choose it as the develop tool is its extensive documentation and great community support. Another reason is that internationally recognized companies including Mozilla, Red Hat, Heroku, and Eventbrite, so it should have a trustworthy reliability. [28]

```
class Search(models.Model):
    #Define fields
    Search_choices = [("All Field", "All Field"), ("Any relationship", "Any relationship"), ("Category", "Category"), ("Child", "Child"),
    search_field = models.CharField(max_length = 100, default = "All Field", choices = Search_choices)
    search_value = models.CharField(max_length = 20, null = True, help_text = "Search Value - Can be any string or 'null' or 'not nu
    show_pro_name = models.BooleanField(default = True)
    show_pro_definition = models.BooleanField(default = True)
    show_category = models.BooleanField(default = True)
    show_parent = models.BooleanField(default = True)
    show_annotation = models.BooleanField(default = False)
    show_relationships = models.BooleanField(default = False)
    show_child = models.BooleanField(default = False)
    show_ecocycid = models.BooleanField(default = False)
    show_genename = models.BooleanField(default = False)
    show_hgncid = models.BooleanField(default = False)
    show_mgiid = models.BooleanField(default = False)
    show_ortho_isoform = models.BooleanField(default = False)
    show_ortho_modified = models.BooleanField(default = False)
    show_pantherid = models.BooleanField(default = False)
    show_pirsfid = models.BooleanField(default = False)
    show_pmid = models.BooleanField(default = False)
    show_reactomeid = models.BooleanField(default = False)
    show_uniprotkbid = models.BooleanField(default = False)
    offset = models.IntegerField(default = 0, help_text="The number of items to skip before starting to collect the result set.")
    limit = models.IntegerField(default = 50, help_text="The numbers of items to return.")
```

Figure 4.3 Python class for search parameters

Figure 4.3 shows the python class for search parameters. There are 22 parameters in this model. The parameter “search\_field” is for the field the search is against. The “search\_value” is the value for the specific field which can be a number

or literal value. When the value of parameter “search\_value” is blank, which is the default value, or not null, it will match any PRO terms with this search field property. If the value is null, the result will return PRO terms without such search field property. The parameter “offset” and “limit” are used for paginating the result to improve performance. The Boolean valued show field parameters simulated the “Display Options” of the PRO text search website. If a show field parameter is set to be True, the value of that field will be included in the returned query result. By default, the name, definition, category and parent of a PRO term are set to be True. The rest of the show field parameters are set to be False.

```
class Retrieval(models.Model):
    proID = models.CharField(max_length = 20)
    show_pro_name = models.BooleanField(default = True)
    show_pro_definition = models.BooleanField(default = True)
    show_category = models.BooleanField(default = True)
    show_parent = models.BooleanField(default = True)
    show_annotation = models.BooleanField(default = False)
    show_relationships = models.BooleanField(default = False)
    show_child = models.BooleanField(default = False)
    show_ecocycid = models.BooleanField(default = False)
    show_genename = models.BooleanField(default = False)
    show_hgncid = models.BooleanField(default = False)
    show_mgiid = models.BooleanField(default = False)
    show_ortho_isoform = models.BooleanField(default = False)
    show_ortho_modified = models.BooleanField(default = False)
    show_pantherid = models.BooleanField(default = False)
    show_pirsfid = models.BooleanField(default = False)
    show_pmid = models.BooleanField(default = False)
    show_reactomeid = models.BooleanField(default = False)
    show_uniprotkbid = models.BooleanField(default = False)
```



Figure 4.4 Python class for retrieval parameters

Figure 4.4 shows the python class for retrieval parameters. In comparison with the class for search parameters, the class for retrieval parameters is simpler. Other than the show field parameters, it only has one parameter, the “proId”. The value of this parameter cannot be blank, so user must input a part or complete valid PRO term ID. It has no search field and search value parameters. This class is mainly for retrieving information about a specific PRO term.

Next important component is Views. Many of the functionalities of the API services are implemented as functions in Views.

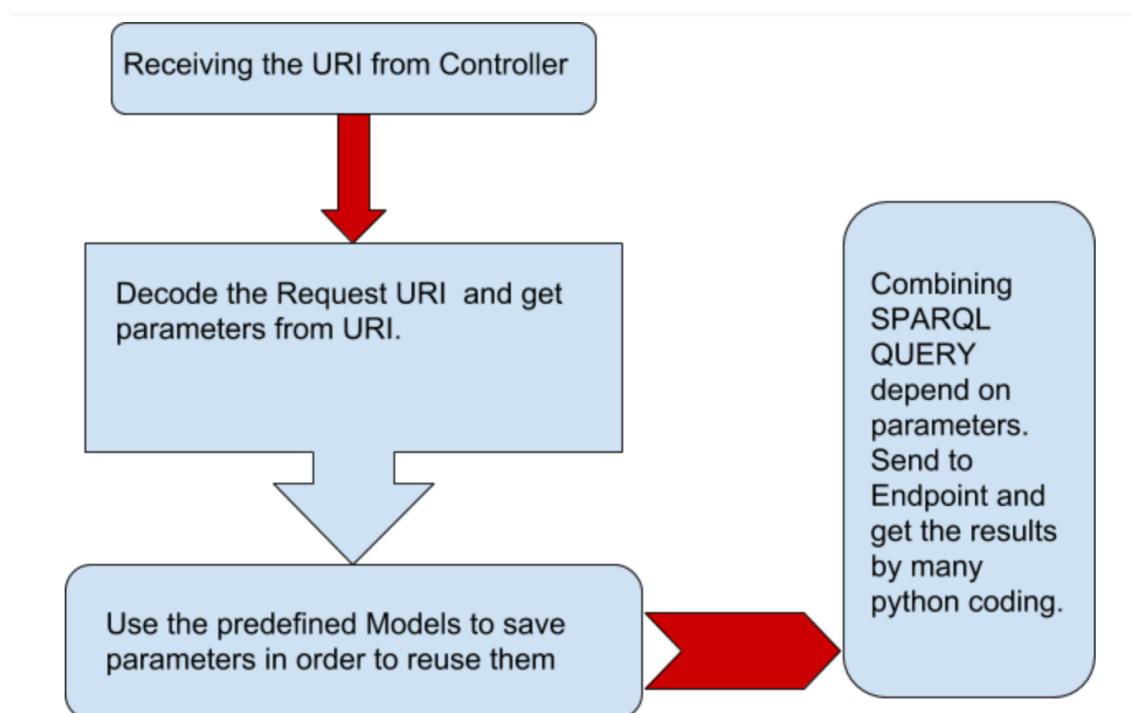


Figure 4.5 Flow chart of function in Views

From the Figure 4.5, there are four parts in the flow chart of Views function. The information encoded in the URL needs to be extracted and processed. As the bridge between other components, there is also a function related to rendering the result to user.

```
@api_view(['GET'])
@renderer_classes((JSONRenderer,XMLRenderer))
def search(request):

    def eqa(qua,res):
        if qua != '':
            if res == "true":
                qua = True
            elif res == "false":
                qua = False
            else:
                qua = int(res)

    inform = request.GET

    argu = defaultdict(lambda:-1)
    for k in inform.iterkeys():
        if k == "searchField" or k == "searchValue":
            argu[k] = inform[k]
        else:
            if inform[k] == "true":
                argu[k] = True
            elif inform[k] == "false":
                argu[k] = False
            else:
                argu[k] = int(inform[k])
    if argu["searchField"] == -1:
        argu["searchField"] = "Any_field"
    a = Search(search_field = argu["searchField"],search_value= argu["searchValue"],
    a.save()
    b = sparqlsearch()
    c = b.search_pre(a)
    d = sort(c)

    return Response(d)
```

Figure 4.6 The layout of function definition in Views

The function starts with some decorators as shown in the yellow box of Figure 4.6. “@api\_view(['GET'])” indicates that it is for handling HTTP GET method request. “@render\_classes ((JSONRenderer, XMLRenderer))” indicates that the HTTP response can be rendered in either JSON or XML format depending on how the Accept format header was set by the user client. By default, JSON response will be generated if user client didn't specify any Accept format.

The code snippets in blue box of Figure 4.6 processes request parameters. There are different types of request parameters. One type of parameters shows searching for a field with specified value. User can use “null” or “not null” as the search field value. Display options parameter determines whether specified field will be queried and become part of the response rendered to the user. So, if a show field is set to be “True” that means the value of that field will be included in the rendered response, “False” means the value of that field will be excluded. Another kind of parameter is used for paginating the result to improve the performance such as “limit” and “offset”. The code snippet in green box does the search against SPARQL endpoint and generates the output.

```

def search_pre(sect,a):
    res = []
    b = application()
    result = defaultdict(lambda:defaultdict(lambda:-1))
    if a.search_field == "Interaction_with" or a.search_field == "Modifier" or a.search_field == "Ontology_ID" or a.search_field == "Ontology_Label":
        anno,annolist = b.search_anno(a,"")
        count = 0
        if a.show_annotation == False:
            for x in annolist:
                main = b.search_main_aft(a,x,"")
                for k in main.iterkeys():
                    for ki in main[k].iterkeys():
                        result[count][ki] = main[k][ki]
                count += 1
            for xi in result.iterkeys():
                res.append(result[xi])
            return res
        else:
            for xi in anno.iterkeys():
                for xo in anno[xi].iterkeys():
                    if "id" in xo:
                        main = b.search_main_aft(a,anno[xi][xo],"")
                        for k in main.iterkeys():
                            for ki in main[k].iterkeys():
                                result[count][ki] = main[k][ki]
                            count += 1
                        result[xi]["annotation"] = anno[xi]
                        res.append(result[xi])
            return res
    else: #The field except these fields in annotation
        main = b.search_main(a,"")
        if a.show_annotation == False:
            for xi in main.iterkeys():
                res.append(main[xi])
            return res
        else:
            for xi in main.iterkeys():
                for xo in main[xi].iterkeys():
                    if "id" in xo:
                        anno,annolist = b.search_anno_aft(a,main[xi][xo],"")
                        an = []
                        for ai in anno.iterkeys():
                            an.append(anno[ai])
                        result[xi]["annotation"] = an
                    else:
                        result[xi][xo] = main[xi][xo]
            res.append(result[xi])
            return res

```

Figure 4.7 Function for constructing SPARQL query dynamically

The search functions in Views are organized into many files according to the search field. In general, they are two steps in the search function. The first step is to construct the SPARQL query based on the input parameter. There are two core functions; one is for searching information in the annotation field, “search\_annotation” and another one is for other fields, “search\_main”. As shown in the Figure 4.7, it has two different conditions. When the field user wants to search is in the field of

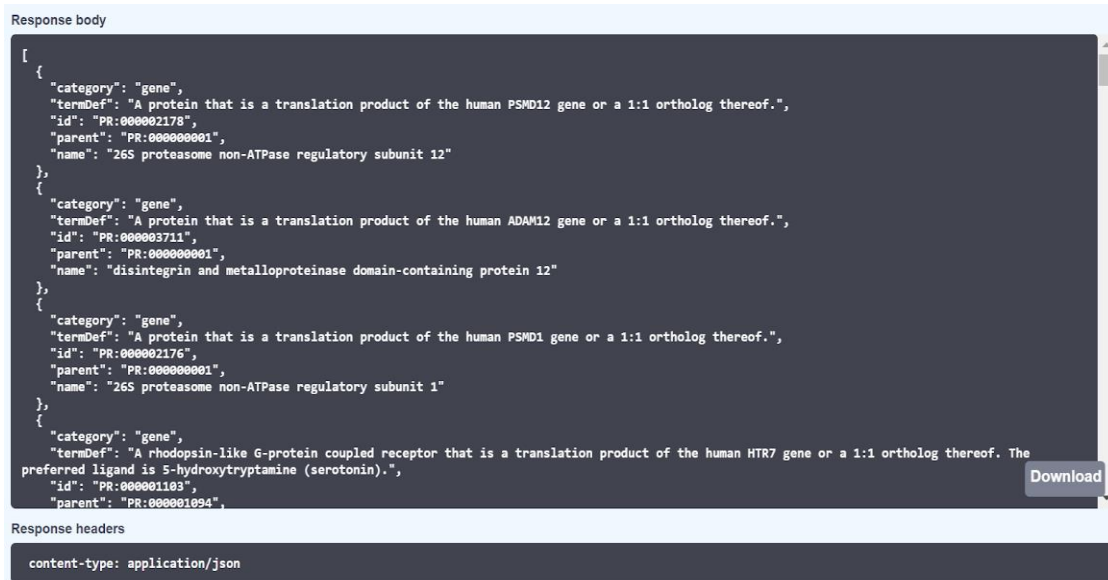
“annotation”, the “search\_annotaion” will be called first to get the information and a list of protein ID based on different conditions will be used to call the function “search\_main” in order to get the complete information about those proteins.

```
urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^pros$', views.search, name = 'search'),
    url(r'^pros/[A-Z0-9:]+$', views.proid, name = 'proid'),
    url(r'^proforms/modification$', views.modification, name = 'modification'),
    url(r'^proforms/modification/phosphorylated$', views.phosphorylated, name = 'phosphorylated'),
    url(r'^proforms/modification/methylated$', views.methylated, name = 'methylated'),
    url(r'^proforms/modification/acetylated$', views.acetylated, name = 'acetylated'),
    url(r'^proforms/modification/ubiquitinated$', views.ubiquitinated, name = 'ubiquitinated'),
    url(r'^proforms/modification/glycosylated$', views.glycosylated, name = 'glycosylated'),
    url(r'^proforms/orthoisoform$', views.orthoisoform, name = 'orthoisoform'),
    url(r'^proforms/orthomodform$', views.orthomodform, name = 'orthomodform'),
    url(r'^proforms/sequence$', views.sequence, name = 'sequence'),
    url(r'^proforms/organism-sequence$', views.organism_sequence, name = 'organism_sequence'),
    url(r'^proevos/family$', views.family, name = 'family'),
    url(r'^proevos/gene$', views.gene, name = 'gene'),
    url(r'^proevos/organism-gene$', views.organism_gene, name = 'organism_gene'),
    url(r'^procomps/species-specific$', views.complex, name = 'complex'),
    url(r'^procomps/species-non-specific$', views.non_complex, name = 'non_complex'),
    url(r'^dbxrefs/EcoCyc_ID$', views.EcoCycID, name = 'EcoCycID'),
    url(r'^dbxrefs/HGNC_ID$', views.HGNCID, name = 'HGNCID'),
    url(r'^dbxrefs/MGI_ID$', views.MGIID, name = 'MGIID'),
    url(r'^dbxrefs/Ontology_ID$', views.OntologyID, name = 'OntologyID'),
    url(r'^dbxrefs/PANTHER_ID$', views.PANTHERID, name = 'PANTHERID'),
    url(r'^dbxrefs/PIRSF_ID$', views.PIRSFID, name = 'PIRSFID'),
    url(r'^dbxrefs/PMID$', views.PMID, name = 'PMID'),
    url(r'^dbxrefs/Reactome_ID$', views.ReactomeID, name = 'ReactomeID'),
    url(r'^dbxrefs/NCBITaxon_ID$', views.TaxonID, name = 'TaxonID'),
    url(r'^dbxrefs/UniProtKB_ID$', views.UniProtKBID, name = 'UniProtKBID'),
    url(r'^paf/[A-Z0-9:]+$', views.annotation, name = 'annotation'),
    url(r'^dag/parent/[A-Z0-9:]+$', views.parent, name = 'parent'),
    url(r'^dag/ancestor/[A-Z0-9:]+$', views.ancestor, name = 'ancestor'),
    url(r'^dag/children/[A-Z0-9:]+$', views.child, name = 'child'),
    url(r'^dag/descendant/[A-Z0-9:]+$', views.descendant, name = 'descendant'),
    url(r'^obo/[A-Z0-9:]+$', views.OBO_Format, name = 'OBO_Format')
]
```

Figure 4.8 URL patterns defined in the Controller

Controller is the entry point in Django REST framework. Each field in the API has their own views function so they also have their own URL patterns. Each line in Figure 4.8 is a URL pattern and must be separated by colon. We use the parameter “name” and views function name to distinguish and identify them. When user enters a

URL into browser, Django will match it against the list. If this pattern is not in the list, the server will return a error to promote user to check the URL.



The screenshot shows a web browser interface with two main sections: 'Response body' and 'Response headers'. The 'Response body' section contains a JSON array of four objects, each representing a gene. The 'Response headers' section shows a single header: 'content-type: application/json'. A 'Download' button is visible on the right side of the JSON array.

```
[
  {
    "category": "gene",
    "termDef": "A protein that is a translation product of the human PSMD12 gene or a 1:1 ortholog thereof.",
    "id": "PR:000002178",
    "parent": "PR:000000001",
    "name": "26S proteasome non-ATPase regulatory subunit 12"
  },
  {
    "category": "gene",
    "termDef": "A protein that is a translation product of the human ADAM12 gene or a 1:1 ortholog thereof.",
    "id": "PR:000003711",
    "parent": "PR:000000001",
    "name": "disintegrin and metalloproteinase domain-containing protein 12"
  },
  {
    "category": "gene",
    "termDef": "A protein that is a translation product of the human PSMD1 gene or a 1:1 ortholog thereof.",
    "id": "PR:000002176",
    "parent": "PR:000000001",
    "name": "26S proteasome non-ATPase regulatory subunit 1"
  },
  {
    "category": "gene",
    "termDef": "A rhodopsin-like G-protein coupled receptor that is a translation product of the human HTR7 gene or a 1:1 ortholog thereof. The preferred ligand is 5-hydroxytryptamine (serotonin).",
    "id": "PR:000001103",
    "parent": "PR:000001094",
    "name": "5-HT2C receptor"
  }
]
```

content-type: application/json

Figure 4.9 An example JSON response from PRO RESTful APIs

The component Template in Django REST framework is just for rendering the output. JSON is a lightweight data exchange format [29], simple and easy to read and write. Figure 4.9 shows an example JSON response from PRO RESTful APIs. Each PRO term is stored in curly braces and square brackets represents the data structure model mentioned above. If user changes the value of “Annotation” from true to false, there will be another curly brace in PRO term. As the key-value pair in the dictionary, the key is the field name and value is the result queried from Virtuoso triple store.

```
Response body
<?xml version="1.0" encoding="utf-8"?>
<root>
  <list-item>
    <category>gene</category>
    <termDef>A protein that is a translation product of the human PSMD12 gene or a 1:1 ortholog thereof.</termDef>
    <id>PR:000002178</id>
    <parent>PR:000000001</parent>
    <name>26S proteasome non-ATPase regulatory subunit 12</name>
  </list-item>
  <list-item>
    <category>gene</category>
    <termDef>A protein that is a translation product of the human ADAM12 gene or a 1:1 ortholog thereof.</termDef>
    <id>PR:000003711</id>
    <parent>PR:000000001</parent>
    <name>disintegrin and metalloproteinase domain-containing protein 12</name>
  </list-item>
  <list-item>
    <category>gene</category>
    <termDef>A protein that is a translation product of the human PSMD1 gene or a 1:1 ortholog thereof.</termDef>
    <id>PR:000002176</id>
    <parent>PR:000000001</parent>
    <name>26S proteasome non-ATPase regulatory subunit 1</name>
  </list-item>
  <list-item>
    <category>gene</category>
    <termDef>A rhodopsin-like G-protein coupled receptor that is a translation product of the human HTR7 gene or a 1:1 ortholog thereof. The preferred ligand is 5-hydroxytryptamine (serotonin).</termDef>
    <id>PR:000001103</id>
  </list-item>
</root>
Download
Response headers
content-type: application/xml; charset=utf-8
```

Figure 4.10 An example XML response from PRO RESTful APIs

XML is a markup language used to encode data or documents [30]. The format of XML has strict standards. The reason PRO RESTful APIs also support rendering XML output is because XML readers and writers have been developed for a variety of programming languages.

### 4.3 Use Case

The REST services provide a flexible interface into multiple aspects of PRO term. In the PRO, the UniProtKB is used to provide the formal definition of protein [31] and there are also some other external databases that are connected to PRO identifiers by the mapping of accessions. RESTful API can help user finding the related information about a protein, like proteoforms, complexes, hierarchy and

annotations. For example, we can get information for the gene level protein class BUB1B (PR:000004855) and its subclasses using APIs.

User can start from “Search PRO terms” API by specifying the “PRO\_term\_definition” field and entering search value “BUB1B” (Figure 4.11) to get a list of PRO terms as shown in Figure 4.12.

The screenshot displays the 'PRO Terms' API interface. At the top, there's a title 'PRO Terms' with a dropdown arrow. Below it, a blue bar contains the HTTP method 'GET' and the endpoint '/pros Search PRO terms.'. A description states: 'Gets a list of PRO terms and associated information.' A 'Parameters' section is highlighted with a red 'Cancel' button. It contains two parameters:

Name	Description
searchField string (query)	Search field that needs to be considered for filter <input type="text" value="PRO_term_definition"/>
searchValue string (query)	any string or "null" or "not null" <input type="text" value="BUB1B"/>

Figure 4.11 Inputs to “Search PRO terms” API.



Curl

```
curl -X GET "https://research.bioinformatics.udel.edu/PRO_API/v1/pros?searchField=PRO_term_definition&searchValue=BUB1&showPROName=true&showPROTermDefinition=true&showCategory=true&showParent=true&showAnnotation=false&showAnyRelationship=false&showChild=false&showEcoCycID=false&showGeneName=false&showHGNCID=false&showMGID=false&showOrthoIsoform=false&showOrthoModifiedForm=false&showPANTHERID=false&showPIRSPID=false&showPMID=false&showReactomeID=false&showUniProtKBID=false&offset=0&limit=50" -H "accept: application/json"
```

Request URL

```
https://research.bioinformatics.udel.edu/PRO_API/v1/pros?searchField=PRO_term_definition&searchValue=BUB1&showPROName=true&showPROTermDefinition=true&showCategory=true&showParent=true&showAnnotation=false&showAnyRelationship=false&showChild=false&showEcoCycID=false&showGeneName=false&showHGNCID=false&showMGID=false&showOrthoIsoform=false&showOrthoModifiedForm=false&showPANTHERID=false&showPIRSPID=false&showPMID=false&showReactomeID=false&showUniProtKBID=false&offset=0&limit=50
```

Server response

Code	Details
200	<p>Response body</p> <pre>{   "id": "PR:000035579",   "name": "BUB1:unphosphoBUB1B complex (human)",   "termDef": "A BUB1:BUB1B complex that contains the unphosphorylated form of BUB1B, and whose components are encoded in the genome of human.",   "category": "Organism-complex",   "parent": "PR:000035578" }, {   "id": "HGNC:1149",   "name": "BUB1B (human)",   "termDef": "A protein coding gene BUB1B in human.",   "category": "external",   "parent": "SO:0001217" }, {   "id": "PR:000004854",   "name": "mitotic checkpoint serine/threonine-protein kinase BUB1",   "termDef": "A BUB1/BUB1B protein that is a translation product of the human BUB1 gene or a 1:1 ortholog thereof.",   "category": "gene",   "parent": "PR:000035665" }, {   "id": "PR:000004855",   "name": "mitotic checkpoint serine/threonine-protein kinase BUB1 beta",   "termDef": "A BUB1/BUB1B protein that is a translation product of the human BUB1B gene or a 1:1 ortholog thereof.",   "category": "gene",   "parent": "PR:000035665" }</pre> <p>Download</p>

Figure 4.12 A list of PRO\_terms returned by “Search PRO terms” API.

From the returned PRO terms, we can see “PR:000004855” has the category of “gene”. We can use “PR:000004855” as input to the “Parent” field of “Organism-gene” API (Figure 4.13) to get a list of organism specific PRO terms as the subclasses of “PR:000004855” (Figure 4.14).

GET /proevos/organism-gene Returns a list of organism-gene level protein terms.

Gets a list of organism-gene level protein terms and associated information.

Parameters

Name	Description
searchField string (query)	Search field that needs to be considered for filter <input type="text" value="Parent"/>
searchValue string (query)	any string or "null" or "not null" <input type="text" value="PR:000004855"/>

Cancel

Figure 4.13 Inputs to “Search organism-gene” API.

Curl

```
curl -X GET "https://research.bioinformatics.udel.edu/PRO_API/V1/provos/organism-gene?searchField=Parent&searchValue=PR:000004855&showPROName=true&showPROTermDefinition=true&showCategory=true&showParent=true&showAnnotation=false&showAnyRelationship=false&showChild=false&showCoCycID=false&showGeneName=false&showHGNCID=false&showMGIID=false&showOrthoIsoform=false&showOrthoModifiedForm=false&showPANTHERID=false&showPINSFID=false&showPKID=false&showReactomeID=false&showUniProtKBID=false&Offset=0&Limit=50" -H "accept: application/json"
```

Request URL

```
https://research.bioinformatics.udel.edu/PRO_API/V1/provos/organism-gene?searchField=Parent&searchValue=PR:000004855&showPROName=true&showPROTermDefinition=true&showCategory=true&showParent=true&showAnnotation=false&showAnyRelationship=false&showChild=false&showCoCycID=false&showGeneName=false&showHGNCID=false&showMGIID=false&showOrthoIsoform=false&showOrthoModifiedForm=false&showPANTHERID=false&showPINSFID=false&showPKID=false&showReactomeID=false&showUniProtKBID=false&Offset=0&Limit=50
```

Server response

Code
Details

200

Response body

```
{
  "name": "mitotic checkpoint serine/threonine-protein kinase BUB1 beta (human)",
  "termDef": "A mitotic checkpoint serine/threonine-protein kinase BUB1 beta that is encoded in the genome of human.",
  "category": "organism-gene",
  "parent": "PR:000004855"
},
{
  "id": "PR:Q8JGT8",
  "name": "mitotic checkpoint serine/threonine-protein kinase BUB1 beta (frog)",
  "termDef": "A mitotic checkpoint serine/threonine-protein kinase BUB1 beta that is encoded in the genome of frog.",
  "category": "organism-gene",
  "parent": "PR:000004855"
},
{
  "id": "PR:Q800D4",
  "name": "mitotic checkpoint serine/threonine-protein kinase BUB1 beta (chicken)",
  "termDef": "A mitotic checkpoint serine/threonine-protein kinase BUB1 beta that is encoded in the genome of chicken.",
  "category": "organism-gene",
  "parent": "PR:000004855"
},
{
  "id": "PR:Q9Z1S0",
  "name": "mitotic checkpoint serine/threonine-protein kinase BUB1 beta (mouse)",
  "termDef": "A mitotic checkpoint serine/threonine-protein kinase BUB1 beta that is encoded in the genome of mouse.",
  "category": "organism-gene",
  "parent": "PR:000004855"
}
]
```

Download

Figure 4.14 A list of organism specific PRO terms as returned by “Search organism-gene” API.

As shown in Figure 4.14, we find 4 organism specific PRO terms that are subclasses of “PR:000004855”: PR:O60566 (human), PR:Q8JGT8 (frog), PR:Q800D4 (chicken) and PR:Q9Z1S0 (mouse). Alternatively, we can also use “Search decedents” API to get all the subclasses of “PR:000004855” and looking for those with category of “organism-gene” as shown in Figure 4.15 and 4.16 Furthermore, user can also get PAF annotation (Figure 4.17 and 4.18).

GET

/dag/descendant/{proId} Returns direct and indirect children PRO terms by the given PRO ID.

Gets direct and indirect children PRO terms by the given PRO ID and associated information.

Parameters

Cancel

Name	Description
<b>proId</b> <span>required</span> string (path)	PRO ID <input type="text" value="PR:000004855"/>

Figure 4.15 Inputs to “Search decedents” API.

Curl

```
curl -X GET "https://research.bioinformatics.udel.edu/PRO_API/V1/dag/descendant/PR:000004855?showPROName=true&showPROTermDefinition=true&showCategory=true&showParent=true&showAnnotation=false&showAnyRelationship=false&showChild=false&showEcoCycID=false&showGeneName=false&showHGNCID=false&showMGIID=false&showOrthoIsoform=false&showOrthoModifiedForm=false&showPANTHERID=false&showPINSID=false&showPMID=false&showReactomeID=false&showUniProtKBID=false&offset=0&limit=50" -H "accept: application/json"
```

Request URL

```
https://research.bioinformatics.udel.edu/PRO_API/V1/dag/descendant/PR:000004855?showPROName=true&showPROTermDefinition=true&showCategory=true&showParent=true&showAnnotation=false&showAnyRelationship=false&showChild=false&showEcoCycID=false&showGeneName=false&showHGNCID=false&showMGIID=false&showOrthoIsoform=false&showOrthoModifiedForm=false&showPANTHERID=false&showPINSID=false&showPMID=false&showReactomeID=false&showUniProtKBID=false&offset=0&limit=50
```

Server response

Code Details

200

Response body

```
{
  "id": "PR:060566",
  "name": "mitotic checkpoint serine/threonine-protein kinase BUB1 beta (human)",
  "termDef": "A mitotic checkpoint serine/threonine-protein kinase BUB1 beta that is encoded in the genome of human.",
  "category": "organism-gene",
  "parent": [
    "PR:000004855"
  ]
},
{
  "id": "PR:Q800D4",
  "name": "mitotic checkpoint serine/threonine-protein kinase BUB1 beta (chicken)",
  "termDef": "A mitotic checkpoint serine/threonine-protein kinase BUB1 beta that is encoded in the genome of chicken.",
  "category": "organism-gene",
  "parent": [
    "PR:000004855"
  ]
},
{
  "id": "PR:Q83078",
  "name": "mitotic checkpoint serine/threonine-protein kinase BUB1 beta (frog)",
  "termDef": "A mitotic checkpoint serine/threonine-protein kinase BUB1 beta that is encoded in the genome of frog.",
  "category": "organism-gene",
  "parent": [
    "PR:000004855"
  ]
}
```

Response headers

Download

Figure 4.16 A list of organism specific PRO terms as returned by “Search decedents” API.

PRO Annotation File

GET /paf/{proId} Returns annotations for the given PRO ID.

Gets annotations for the given PRO ID.

Parameters

Cancel

Name	Description
proId <span style="color: red;">*</span> required	PRO ID
string (path)	PR:000035430

Execute Clear

Figure 4.17 Inputs to “Get PAF annotation” API.

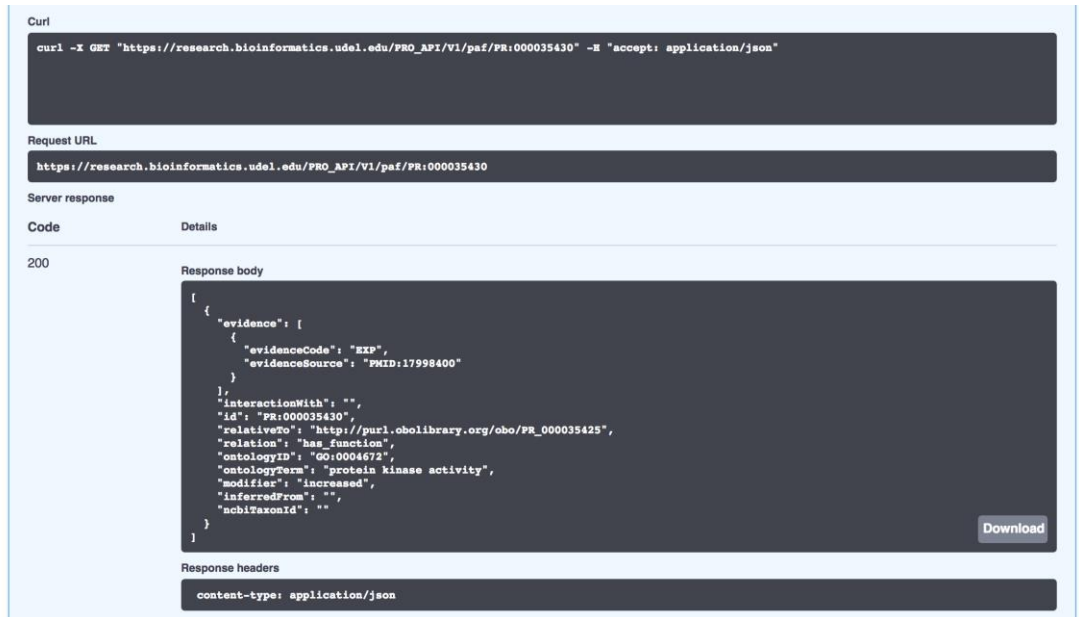


Figure 4.18 Annotations returned by “Get PAF annotation” API.

Other than Swagger UI web interface and Curl command, the PRO RESTful APIs can be invoked by a programming language and executed. Figure 4.19 shows a Python script executes the search of a PRO term by its id.

```
import requests, sys

requestURL
="https://research.bioinformatics.udel.edu/PRO_API/V1/pros/PR:000004855?showPRName=true" \
"&showPROTermDefinition=true&showCategory=true&showParent=true&showAnnotation=false" \
"&showAnyRelationship=false&showChild=false&showEcoCycID=false&showGeneName=false" \
"&showHGNCID=false&showMGIIID=false&showOrthoIsoform=false&showOrthoModifiedForm=false" \
"&showPANTHERID=false&showPIRSFID=false&showPMID=false&showReactomeID=false&showUniProtKBID=
false"

r = requests.get(requestURL, headers={"Accept":"application/json"})

if not r.ok:
    r.raise_for_status()
    sys.exit()

responseBody = r.text

print(responseBody)
```

Figure 4.19. Python script executes the search of a PRO term by its id.

## Chapter 5

### DISCUSSION AND FUTURE WORK

The integration of heterogeneous data can significantly reduce the difficulty of database's maintenance. In the original version of PRO database, four different types of databases are used: SQL-Lite, Oracle, Apache Lucene and Virtuoso. Each database needs a series of processes to maintenance like checking the schema, models and data files. The DBAs need to check and update four databases at the same time in order to make PRO work regularly and smoothly. This often made the DBAs stressful. In current version of PRO database, there is only one Virtuoso database server. There is no doubt that the workload for DBA is reduced significantly.

Furthermore, the simplification of databases and data structure can also improve the query efficiency. As indicated in the performance evaluation presented in Chapter 3, the new Virtuoso/SPARQL based search engine has comparable performance with respect to Apache Lucene based search engine. For some queries, the new one is significant better. The stability of new searching engine is also better than that of old engine. In addition, the integrity of data is also improved to some extent. Finally, the PRO RESTful APIs provide programmatic access to PRO database that can help bioinformatics developer build novel application to use PRO data.

However, there are still some improvements. For example, the SPARQL query can be further optimized. The SPARQL query library can also be improved. The PRO entry page and visualization website [32] is still using Oracle database as backend. It can be modified to use the PRO RESTful APIs as the backend instead. In addition, as

the database evolves, more data and fields will be introduced, the SPARQL query library and RESTful APIs also need to be updated in the future.

## **Chapter 6**

### **CONCLUSION**

In conclusion, the semantic web technologies such as RDF and SPARQL etc. are suitable for data integration. By using RDF, the data is structured and simplified. Compared to unstructured data, the structured data has a strict standard format and can simplify the query process and improve efficiency. At the same time, expandability and flexibility of data are also significantly improved so that we can store data at any time without having to create new field in the SQL table. This is especially important for dealing with Big Data.

The thesis presents the integration of heterogeneous data using semantic web technologies. In addition, it also showed the design and implementation of the RESTful APIs in detail along with application examples. The thesis aims to provide a clear description of the heterogeneous data integration process and API construction process. The thesis can also be used as a reference for API development in the field of Bioinformatics.

The Virtuoso/SPARQL powered PRO text search website [33] and the API documentation website [34] are accessible by the URLs in the References.

This thesis work has been presented in the 7th Annual Big Data in Biomedicine Symposium held in Georgetown University, Washington DC on October 26, 2018.

## REFERENCES

1. Michelle Cheatham, Catia Pesquita. (2017). The semantic Data integration. A.Y. Zomaya and S. Sakr (eds.), *Handbook of Big Data Technologies*, Springer International Publishing
2. Berners-Lee, Tim (May 17, 2001). "The Semantic Web" (PDF). *Scientific American*. Retrieved October 26, 2018, from <https://pdfs.semanticscholar.org/566c/1c6bd366b4c9e07fc37eb372771690d5ba31.pdf>
3. Tom Heath, Christian Bizer, Synthesis Lectures. (2011) *Linked Data: Evolving the Web into a Global Data Space*. Retrieved from <http://linkeddatabook.com/book>
4. Ahmet Soylu, Felix Mödritscher, and Patrick De Causmaecker. 2012. "Ubiquitous Web Navigation through Harvesting Embedded Semantic Data: A Mobile Scenario." *Integrated Computer-Aided Engineering* 19 (1): 93–109.
5. Semantic Web Architecture.(2007).Retrieved October 14, 2018, from : <http://obitko.com/tutorials/ontologies-semantic-web/semantic-web-architecture.html>.
6. W3C. RDF - Semantic Web Standards. <https://www.w3.org/RDF/>
7. B. McBride, The Resource Description Framework (RDF) and its Vocabulary Description Language RDFS, in:*The Handbook on Ontologies in Information Systems*, S. Staab,R.Studer(eds.),Springer Verlag,2003.
8. "XML and Semantic Web W3C Standards Timeline" (PDF). 2012-02-04. Retrieved October 26, 2018, from <http://www.dblab.ntua.gr/~bikakis/XMLSemanticWebW3CTimeline.pdf> .
9. W3C. SPARQL 1.1(2013, March 21). Retrieved October 26, 2018, from <https://www.w3.org/TR/sparql11-overview/>
10. Segaran, Toby; Evans, Colin; Taylor, Jamie (2009). Programming the Semantic Web. *O'Reilly Media, Inc.*, p. 84. ISBN 978-0-596-15381-6.



11. Jim Rapoza (2006, May 2). "SPARQL Will Make the Web Shine". *eWeek*. Retrieved October 26, 2018, from <http://www.eweek.com/development/sparql-will-make-the-web-shine> .
12. Nadine Schuurman, Agnieszka Leszczynski (2008). Ontologies for Bioinformatics. *Bioinform Biol Insights*. 2008; 2: 187–200.
13. Natale DA, Arighi CN, Blake JA, Bona J, Chen C, Chen SC, Christie KR, Cowart J, D'Eustachio P, Diehl AD, Drabkin HJ, Duncan WD, Huang H, Ren J, Ross K, Ruttenberg A, Shamovsky V, Smith B, Wang Q, Zhang J, El-Sayed A, Wu CH. "Protein Ontology (PRO): enhancing and scaling up the representation of protein entities." *Nucleic Acids Res*. 2017 Jan 4;45(D1): D339-D346. doi: 10.1093/nar/gkw1075. Epub 2016 Nov 28.
14. "Framework Figure". (PDF) (2010, April). Retrieved October 26, 2018, from [https://pir.georgetown.edu/pro/documents/framework\\_figure.pdf](https://pir.georgetown.edu/pro/documents/framework_figure.pdf).
15. W3C Semantic Web Activity". *World Wide Web Consortium (W3C)*. November 7, 2011. Retrieved October 26, 2018.
16. Fielding, Roy Thomas. Chapter 5: Representational State Transfer (REST). Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine. 2000.
17. Erl, Thomas; Carlyle, Benjamin; Pautasso, Cesare; Balasubramanian, Raj (2012). "5.1". SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST. Upper Saddle River, New Jersey: Prentice Hall. ISBN 978-0-13-701251-0.
18. Sayers E. A General Introduction to the E-utilities. In: *Entrez Programming Utilities Help* [Internet]. Bethesda (MD): National Center for Biotechnology Information (US); 2010-. Available from: <https://www.ncbi.nlm.nih.gov/books/NBK25497/>
19. Veronique Greenwood. (2016, April 26). *Life's Blueprints*. Retrieval October 26, 2018, from <https://www.quantamagazine.org/one-gene-many-proteins-20160426/>
20. Linux Foundation wants to extend Swagger in connected buildings". (2015, November 6). Retrieval October 26, 2018, from <http://www.businesscloudnews.com/2015/11/06/linux-foundation-wants-to-extend-swagger-in-connected-buildings/> .
21. Adrian Holovaty, Jacob Kaplan-Moss; et al. *The Django Book*.

22. M. Lenzerini. Data integration: a theoretical perspective. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 233–246, 2002.
23. "RDF 1.1 Turtle - Terse RDF Triple Language Turtle". *World Wide Web Consortium (W3C)*. (2014, February 25). Retrieved October 26, 2018, from <http://www.w3.org/TR/turtle/> .
24. "Lucene Implementations". *apache.org*. Archived from the original on 6 October 2015. Retrieved October 26, 2018, from <https://web.archive.org/web/20151006021755/http://wiki.apache.org/lucene-java/LuceneImplementations> .
25. 25. ALISDAIR OWENS, “An Investigation into Improving RDF Store Performance an Investigation into Improving RDF Store Performance”, PHD Thesis, UNIVERSITY OF SOUTHAMPTON, 2009.
26. <https://pir17.georgetown.edu/confluence/display/PROWIKI/PRO+Ontology+Manual+Curation+Guideline>
27. [https://beadle.dbi.udel.edu/pro/pro\\_api.shtml](https://beadle.dbi.udel.edu/pro/pro_api.shtml)
28. Django REST Framework (2011) Retrieved October 26, 2018, from <https://www.django-rest-framework.org/#> .
29. Introducing JSON. Retrieved October 26, 2018, from <http://json.org/> .
30. Bikakis N, Tsinaraki C, Gioldasis N, Stavrakantonakis I, Christodoulakis S. "The XML and Semantic Web Worlds: Technologies, Interoperability and Integration. A survey of the State of the Art" In *Semantic Hyper/Multimedia Adaptation: Schemes and Applications*, Springer 2013.
31. atale DA, Arighi CN, Blake JA, Bona J, Chen C, Chen SC, Christie KR, Cowart J, D'Eustachio P, Diehl AD, Drabkin HJ, Duncan WD, Huang H, Ren J, Ross K, Ruttenberg A, Shamovsky V, Smith B, Wang Q, Zhang J, El-Sayed A, Wu CH. "Protein Ontology (PRO): enhancing and scaling up the representation of protein entities." *Nucleic Acids Res*. 2017 Jan 4;45(D1): D339-D346. doi: 10.1093/nar/gkw1075. Epub 2016 Nov 28.

32. Natale DA, Arighi CN, Blake JA, Bult CJ, Christie KR, Cowart J, D'Eustachio P, Diehl AD, Drabkin HJ, Helfer O, Huang H, Masci AM, Ren J, Roberts NV, Ross K, Ruttenberg A, Shamovsky V, Smith B, Yerramalla MS, Zhang J, AlJanahi A, Celen I, Gan C, Lv M, Schuster-Lezell E, Wu CH. "Protein Ontology: a controlled structured network of protein entities." *Nucleic Acids Res.* 2014, 42(Database issue): D415-21. doi: 10.1093/nar/gkt1173.
33. The PRO entry page and visualization website, Retrieved October 26, 2018, from [https://research.bioinformatics.udel.edu/pro/entry/\[PRO\\_ID\]/](https://research.bioinformatics.udel.edu/pro/entry/[PRO_ID]/)
34. PRO search website, Retrieved October 26, 2018, from [https://beadle.dbi.udel.edu/cgi-bin/pro/textsearch\\_sparql?search=1](https://beadle.dbi.udel.edu/cgi-bin/pro/textsearch_sparql?search=1)
35. The API documentation website, Retrieved October 26, 2018, from [https://beadle.dbi.udel.edu/pro/pro\\_api.shtml](https://beadle.dbi.udel.edu/pro/pro_api.shtml)

## Appendix A

### COMPARSION OF PERFORMANCE IN NEW/OLD DATABASE

SEARCH FIELD	SEARCH ENGINE	MINIMUM TIME(s)	MAXIMUM TIME(s)	AVERAGE TIME(s)	STANDARD DEVITATION
Any Fields	Virtuoso/SPAR QL	2.277	2.354	2.302	0.010
	Apache Lucene	1.551	5.344	2.386	0.988
Comment	Virtuoso/SPAR QL	0.902	1.048	0.928	0.043
	Apache Lucene	0.252	1.329	0.581	0.495
Modifier	Virtuoso/SPAR QL	0.277	0.313	0.296	0.037
	Apache Lucene	0.296	1.368	0.777	0.511
Any_relationship	Virtuoso/SPAR QL	1.494	1.721	1.538	0.066
	Apache Lucene	0.460	2.959	1.505	0.884
Gene name	Virtuoso/SPAR QL	1.111	1.158	1.127	0.013
	Apache Lucene	0.302	2.707	1.029	0.875
Ecocyc ID	Virtuoso/SPAR QL	0.168	0.205	0.183	0.013
	Apache Lucene	0.202	0.221	0.211	0.007

Parent	Virtuoso/SPAR QL	1.881	1.893	1.888	0.004
	Apache Lucene	0.390	2.866	1.041	0.825
Ontology ID	Virtuoso/SPAR QL	0.305	0.352	0.331	0.014
	Apache Lucene	0.278	1.369	0.864	0.403
Iso_form	Virtuoso/SPAR QL	0.243	0.282	0.257	0.011
	Apache Lucene	0.236	2.004	0.627	0.627
Modified_form	Virtuoso/SPAR QL	0.180	0.234	0.202	0.015
	Apache Lucene	0.184	0.278	0.1997	0.029

**Appendix B**

**CATEGORY OF PRO API**

<b>API OPERATION GROUP</b>	<b>PATH</b>	<b>DESCRIPTION</b>
<b>PRO TERMS</b>	/pros	Gets a list of PRO terms and associated information.
	/pros/{proID}	Gets one or more PRO terms and associated information. by ID.
<b>Proteoform Terms</b>	/proforms/modification	Gets a list of modified protein forms and associated information.
	/proforms/modification/phosphorylated	Gets a list of phosphorylated protein forms and associated information.
	/proforms/modification/methylated	Gets a list of methylated protein forms and associated information.
	/proforms/modification/acetylated	Gets a list of acetylated protein forms and associated information.
	/proforms/modification/ubiquitinated	Gets a list of ubiquitinated protein forms and associated information.
	/proforms/modification/glycosylated	Gets a list of glycosylated protein forms and associated information.

	/proforms/orthoisoform	Gets a list of ortho-isoform protein forms and associated information.
	/proforms/orthomodform	Gets a list of ortho-modform protein forms and associated information.
	/proforms/sequence	Gets a list of sequence level protein forms and associated information.
	/proforms/organism-sequence	Gets a list of organism-sequence level protein forms and associated information.
<b>Protein Evolutionary Terms</b>	/proevos/family	Gets a list of family level protein terms and associated information.
	/proevos/gene	Gets a list of gene level protein terms and associated information.
	/proevos/organism-gene	Gets a list of organism-gene level protein terms and associated information.
<b>Protein Complex Terms</b>	/procomps/species-specific	Gets a list of species specific protein complex terms and associated information.
	/procomps/species-non-specific	Gets a list of species non-specific protein complex terms and associated information.
<b>Database Cross-references</b>	/dbxrefs/EcoCyc_ID	Gets a list of PRO terms with EcoCyc ID as cross-reference and associated information.
	/dbxrefs/HGNC_ID	Gets a list of PRO terms with HGNC ID as cross-reference

		and associated information.
	/dbxrefs/MGI_ID	Gets a list of PRO terms with MGI ID as cross-reference and associated information.
	/dbxrefs/Ontology_ID	Gets a list of PRO terms with Ontology ID as cross-reference and associated information.
	/dbxrefs/PANTHER_ID	Gets a list of PRO terms with PANTHER ID as cross-reference and associated information.
	/dbxrefs/PIRSF_ID	Gets a list of PRO terms with PIRSF ID as cross-reference and associated information.
	/dbxrefs/PMID	Gets a list of PRO terms with PMID as cross-reference and associated information.
	/dbxrefs/Reactome_ID	Gets a list of PRO terms with Reactome ID as cross-reference and associated information.
	/dbxrefs/NCBITaxon_ID	Gets a list of PRO terms with NCBI Taxon ID as cross-reference and associated information.
	/dbxrefs/UniProtKB_ID	Gets a list of PRO terms with UniProtKB ID as cross-reference and associated information.
<b>PRO Annotation File</b>	/paf/{proId}	Gets annotations for the given PRO ID.



<b>OBO File</b>	/obo/{proId}	Gets PRO term in OBO format for the given PRO ID.
<b>Hierarchy</b>	/dag/parent/{proId}	Gets direct parent PRO terms by the given PRO ID and associated information.
	/dag/ancestor/{proId}	Gets direct and indirect parent PRO terms by the given PRO ID and associated information.
	/dag/children/{proId}	Gets direct children PRO terms by the given PRO ID and associated information.
	/dag/descendant/{proId}	Gets direct and indirect children PRO terms by the given PRO ID and associated information.